# Algorithmic Advances in Handling Uncertainty & Regularity in Strings

Kundu, Ritu

*Awarding institution:*
King's College London

# Algorithmic Advances in Handling Uncertainty & Regularity in Strings

*- for Genomic Data Analysis -*

By

RITU KUNDU


Department of Informatics
KING'S COLLEGE LONDON

A dissertation submitted for the degree of DOCTOR OF PHILOSOPHY.

JULY 2018

# Abstract

Genomics, owing to its immediate applications in medicine, forensics, evolutionary and molecular biology etc., has witnessed a dramatic advancement in the technology of acquiring and generating data. Consequently, the bottleneck of the information-extraction pipeline has shifted from data-acquisition to the computational capacity for storing and processing prodigious amounts of data. *Uncertainty* and identification of *regularity* in data are two key aspects contributing to the complexity of the task of mining knowledge and insights from genomic data.

One form of macro-level uncertainty arises in sequential data when a single representation is used for a multitude of strings which are by and large similar. For example, in human genomics, the reference genome has been represented as a single sequence so far. Now, with the availability of a vast collection of human genomes, the so called *reference cohorts* seem more sensible in order to avoid the reference-bias presented by a single genomic sequence. Different representations have recently been explored in an attempt to organise human genomic sequences in reference cohorts. Each such representation has its own challenges.

Moreover, in genomic sequences, local regularity (a term encapsulating various forms of repetitions) is often flanked by regions of interest – genes, for example – which are, in comparison, not regular. In other words, the regularity of a local segment of genomic data is indicative of it being a potential *biologically-important region*. One of the multiple possible ways to express this notion of local regularity of strings can be in terms of *unbordered factors* of a string. A *border* of a string – one of the central properties characterising the regularity associated with repetitions – is its (possibly empty) proper factor occurring both as a prefix and as a suffix.

This dissertation presents an assortment of efficient novel algorithms – based on string algorithms and data-structures – to solve three problems that find direct or indirect applications in genomic data analysis. Specifically, the presented algorithms handle the uncertainty arising in the representation of an ensemble of sequences as well as characterise the regularity present in a sequence in terms of unbordered factors.

Firstly, we present an optimal algorithm – in terms of both time and space – improving the state-of-the-art, to identify **Superbubbles** (a special type of self-contained subgraphs, each with a single source and a single sink) in de Bruijn sequence graphs for genome assembly. Identifying these motifs in a reference graph is crucial for overcoming the lack of a coordinate system in the graphical representa-

tion of a reference cohort.

Secondly, we introduce another representation for sequential data with macro-level uncertainty, called **Elastic-degenerate strings**. The motivation is to condense a set of genomes (with variations) as a reference cohort. An *elastic-degenerate string* is a string in which an *elastic-degenerate symbol* can occur at one or more positions; each such symbol corresponds to a set of two or more variable-length strings. We not only formalise the concept of elastic-degenerate strings but also present a practically efficient algorithm to solve the pattern matching problem in a given elastic-degenerate text.

Lastly, we provide a quasilinear time algorithm to compute the **Longest Unbordered Factor Array** of a string $w$ for general alphabets. This array specifies the length of the *maximal unbordered factor* (the longest factor which does not have a border) starting at each position of $w$. This is a major improvement on the running time of the currently best worst-case algorithm working in $\mathcal{O}(n^{1.5})$ time for integer alphabets, where $n$ is the length of $w$. Although this problem is rooted in theory, the data-structures proposed in this algorithm can be used to characterise the regularity of a sequence; this has possible applications in genomics.

# Dedication and acknowledgements

As this dissertation sums up my academic journey of the past three years, I would like to take this opportunity to express my heartfelt gratitude to all the people who made it possible.

First and foremost, I would like to thank my supervisors, **Dr. Toktam Mahmoodi** and **Dr. Solon Pissis**, whose constant support and guidance made me sail smoothly through to the end. Like ideal teachers, their supervision went beyond research and career related advice – they ensured that I successfully overcome every professional hurdle that I encountered. I aspire to emulate their zeal and sincerity as scientists as well as their impeccable work ethics.

My sincerest thanks to **Prof. Maxime Crochemore** – I have learnt a great deal from his insights on a wide range of disparate subjects concerning Algorithmics, Science, World, Food, Life and so on. No amount of thanks will suffice for my colleagues and friends, **Dr. Manal Mohamed** and **Panagiotis Charalampopoulos** (Panos). I feel that working with Manal has notably improved my scientific aptitude and strengthened my technical foundations. More importantly, being a brilliant scientist and a kind-hearted and affable human being, she has been a source of inspiration. Similarly, Panos, due to his academic excellence and warm nature, has always made our brainstorming sessions intellectually stimulating and pleasant. I deeply appreciate the companionship of Manal and Panos through all my personal and professional ups and downs.

My cordial thanks to **Prof. Thomas Erlebach** and **Prof. Leszek Gąsieniec** for agreeing to examine this dissertation and for their valuable suggestions and corrections to improve the dissertation.

Special acknowledgement is due to the **Department of Informatics** and its **professional services staff** for their proficient assistance in the administrative matters. I earnestly acknowledge the support provided by the **Engineering and Physical Sciences Research Council (EPSRC)** in the form of studentship.

All the collaborators and colleagues deserve a special thanks who have contributed positively to my learning curve – **Dr. Simon Puglisi**, **Dr. Golnaz Badkobeh**, **Dr. Robert Mercas**, **Dr. Sharma Thankachan**, **Fatima Vayani**, **Steven Watts**, **Lorraine Ayad** to name a few.

Last but not the least, I am most grateful to my family – my parents (**Smt. Anita & Sh. Ramesh Kundu**), my siblings (**Swati and Mohit**), my parents-in-law (**Smt. Rama & Sh. Raj Kumar Sidhu**), and my husband (**Rahul**). My family is the

bedrock of my life. Swati, Mohit, and Rahul are my strength and the driving force behind my aspirations. I dedicate this and every milestone that I may achieve in future to my family.

## AUTHOR'S DECLARATION

I declare that the work in this dissertation was carried out in accordance with the requirements of the University's Regulations and Code of Practice for Research Degree Programmes and that it has not been submitted for any other academic award. Except where indicated by specific reference in the text, the work is the candidate's own work. Work done in collaboration with, or with the assistance of, others, is indicated as such. Any views expressed in the dissertation are those of the author.

RITU KUNDU

JULY 09, 2018

# Table of Contents

# List of Publications

The author has contributed to the following publications[i] during the course of completion of this dissertation. However, only the first three (marked with ■) constitute the basis of the dissertation. For each of the rest of the publications, the author's reason for its exclusion is either that its subject-area is unrelated to the theme of this dissertation or that the author's contribution is not significant enough to claim the ownership of its work.

1. ■ L. Brankovic, C. S. Iliopoulos, **R. Kundu**, M. Mohamed, S. P. Pissis, F. Vayani, "Linear-Time Superbubble Identification Algorithm for Genome Assembly", *Theoretical Computer Science*, vol. 609, Part 2, 2016, pp. 374–383 [BIK+16].

2. ■ C. S. Iliopoulos, **R. Kundu** and S. P. Pissis, "Efficient Pattern Matching in Elastic-Degenerate Texts", in *Language and Automata Theory and Applications: 11th International Conference (LATA) Proceedings*, Springer, 2017, pp. 131–142 [IKP17].

3. ■ T. Kociumaka, R. Kundu, M. Mohamed, S. P. Pissis, "Longest Unbordered Factor in Quasilinear Time", in *29th International Symposium on Algorithms and Computation (ISAAC 2018)*, W. Hsu, D. Lee, C. Liao, Eds., Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, pp. 70:1–70:13 [KKMP18].

4. M. Crochemore, C. S. Iliopoulos, **R. Kundu**, M. Mohamed, F. Vayani, "Linear algorithm for conservative degenerate pattern matching", *Engineering Applications of Artificial Intelligence*, vol. 51, 2016, pp. 109–114 [CIK+16b].

[i] The norm in the author's research-field is to order the names of the co-authors lexicographically based on their last names.

5. **R. Kundu**, T. Mahmoodi, "Mining Acute Stroke Patients' Data using Supervised Machine Learning", in *Mathematical Aspects of Computer and Information Sciences: 7th International Conference, MACIS 2017, Proceedings*, Springer, 2017, pp. 364–377 [KM17].

6. C. S. Iliopoulos, **R. Kundu** and M. Mohamed, "Efficient Computation of Clustered-Clumps in Degenerate Strings", in *Artificial Intelligence Applications and Innovations (AIAI) Proceedings*, Springer, 2016, pp. 510–519 [IKM16].

7. C. S. Iliopoulos, **R. Kundu**, M. Mohamed, F. Vayani, "Popping Superbubbles and Discovering Clumps: Recent Developments in Biological Sequence Analysis", in *Algorithms and Computation: 10th International Workshop (WALCOM) Proceedings*, Springer, 2016, pp. 3–14 [IKMV16].

8. M. Crochemore, C. S. Iliopoulos, T. Kociumaka, **R. Kundu**, S. P. Pissis, J. Radoszewski, W. Rytter, T. Walen, "Near-Optimal Computation of Runs over General Alphabet via Non-Crossing LCE Queries", in *International Symposium on String Processing and Information Retrieval (SPIRE)*, Springer, 2016, pp. 22–34 [CIK$^+$16a].

9. C. Barton, C. S. Iliopoulos, **R. Kundu**, S. P. Pissis, A. Retha, F. Vayani, "Accurate and efficient methods to improve multiple circular sequence alignment", in *Experimental Algorithms: 14th International Symposium, (SEA) Proceedings*, Springer, 2015, pp. 247–258 [BIK$^+$15].

10. I. Fontana, G. Giacalone, A. Bonanno, S. Mazzola, G. Basilone, S. Genovese, S. Aronica, S. Pissis, C. S. Iliopoulos, **R. Kundu**, A. Fiannaca, A. Langiu, G. L. Bosco, M. L. Rosa, R. Rizzo, "Pelagic Species Identification by using a Probabilistic Neural Network and Echo-sounder Data", in *Artificial Neural Networks and Machine Learning – ICANN 2017: 26th International Conference on Artificial Neural Networks, Proceedings*, Springer, Part 1, 2017, pp. 454–455 [FGB$^+$17].

11. A. Bhardwaj, B. Cizmeci, E. Steinbach, Q. Liu, M. Eid, J. AraUjo, A. E. Saddik, **R. Kundu**, X. Liu, O. Holland, M. A. Luden, S. Oteafy, V. Prasad, "A candidate hardware and software reference setup for kinesthetic codec standardization", in 2017 *IEEE International Symposium on Haptic, Audio and Visual Environments and Games (HAVE)*, pp. 1–6 [BCS$^+$17].

# List of Software

The author has developed the following related software-tools during the course of completion of this dissertation. These tools are open source and are freely available on GitHub Platform [ii].

- **SUPBUB: Superbubbles**: A tool that, in linear time, finds out superbubbles (special graphical motifs) in a directed graph. It is being used in the vg (variation graph) toolkit developed by *Richard Durbin's lab at the Wellcome Trust Sanger Institute*.
- **ElDeS: Elastic-Degenerate Strings**: A tool that finds out occurrences of degenerate patterns in an elastic-degenerate text.
- **luf: Longest Unbordered Factor Array**: A tool that computes the Longest Unbordered Factor Array of a string in quasi-linear time.
- **Clustered Clumps**: A tool that finds out clustered-clumps of – degenerate patterns in a solid text; solid patterns in a degenerate text; and degenerate patterns in a degenerate text.
- **APDS: Approximate Pattern-matching in Degenerate Strings**: A tool that finds out (in linear time) approximate occurrences of a degenerate pattern in a given text sequence.

[ii] https://github.com/Ritu-Kundu

# List of Algorithms

# List of Figures

CHAPTER 1

# INTRODUCTION

Storing, processing, and analysing data are the fundamental stages in the pipeline of extracting usable information from raw data. The more voluminous the data, the more reliable is the extracted information, but also more computationally challenging is its processing and analysis. Although computational power has risen sharply over the last two decades, its growth has not been able to keep pace with the explosive rate at which data is being generated. As a result, the efficiency of algorithmic tools plays a crucial role in providing viable solutions to various problems and issues encountered during a meaningful interpretation of data.

Computational biology is one of the prominent fields to have witnessed a dramatic advancement in technology. This has shifted the bottleneck of the information-extraction pipeline from data-acquisition to the computational capacity for storing and analysing the prodigious amounts of data. Genomics, in particular, is a case in point. A genome is the complete set of DNA (Deoxyribonucleic Acid) of an organism for most species or of RNA (Ribonucleic acid) in some viruses. Genomics, being the branch of molecular biology focussing on the structure, function, evolution, mapping etc. of genomes, entails sequencing, assembling, and analysis of genomes. Genome sequencing technology, which discerns the order of nucleotides making up an organism's DNA, has progressed significantly from the initial sequencers developed about 40 years back [MG77, SNC77] to the current state-of-the-art "high-throughput" (formerly, "next generation") sequencers (for an overview of sequencing technolo-

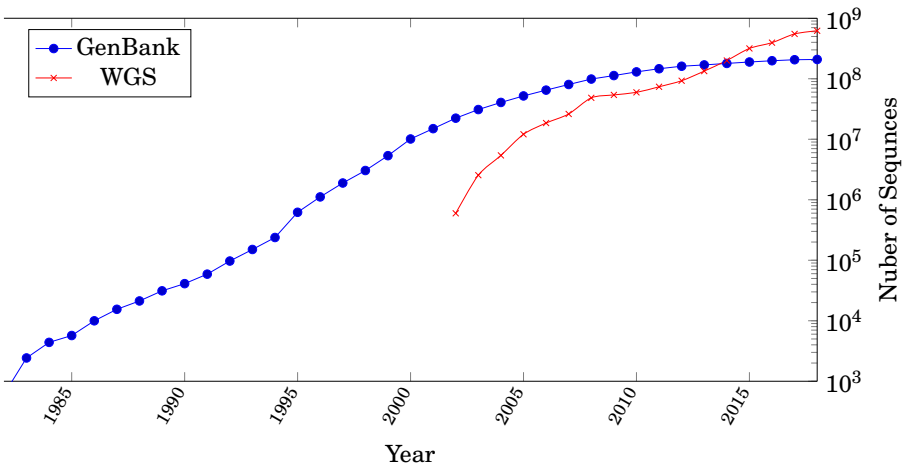


Figure 1.1: Plot showing the exponential growth of number of sequences in the GenBank and WGS databases.

gies, next generation sequencers, and their applications, see [Cha05, BdD14]). As a concomitant effect, the rate at which sequences are added to the databases like *GenBank* [CKML+16] over the past one and a half decade has been explosive; Figure 1.1 demonstrates the number of sequences submitted to GenBank including bulk submissions of whole genome shotgun (WGS) projects (statistics taken from the GenBank website [GS18]).

Owing to the immediate applications of genomics in medicine, forensics, evolutionary and molecular biology etc., genome sequencing technology continues to improve. Consequently, third generation sequencers like the Nanopore sequencer [LGN16] have already emerged which makes sequencing possible at a rate that was unimaginable before. For coping with this massive scale of production of genomic sequences, it has become vital to develop new and improved algorithmic techniques and tools which can analyse and interpret this sequential data as fast and as efficiently as possible.

The increased volume of the data is not the only emerging new challenge. *Uncertainty* in the data is another and so is the need for identifying characteristic *regularity* in it. Figure 1.2 delineates the general challenges in the analysis of genomic data. The challenging dimensions of uncertainty and identifying regularity in data have been elucidated in the following two sections.

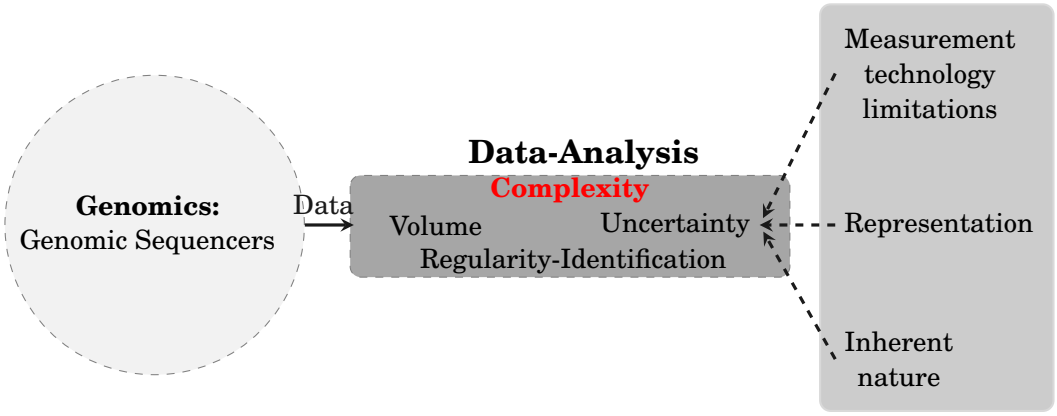


Figure 1.2: Figure representing complexity in genomic data-analysis.

## 1.1 Uncertainty

Genomic data (DNA), at the most basic level, is structured as a sequence of repeating subunits called nucleotides which are identified by four bases – adenine (A), cytosine (C), guanine (G) and thymine (T). In other words, genomic sequences are *strings* over the fixed *alphabet* (represented as $\Sigma$) consisting of $\{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}\}$. Uncertainty in genomic data is usually a causal effect of one or more of the following:

- **Limitations of measurement technology:** Inaccuracies or discrepancies in data lead to uncertainty. Inaccuracies can be introduced in data by the errors made during its collection or generation. For example, genome sequencers are inherently inaccurate, resulting in erroneous and spurious readings in determining which of the four bases occurs at a particular position.
- **Data-representation:** Modelling of the stored data in order to address specific concerns during its processing or analysis can give rise to uncertainty. For instance, succinctly representing multiple similar genomes as one can cause simple sequential raw data to transform into a form where multiple subsequences can occur at the same position.
- **Inherent nature of data:** Genetic mutations and repeats in the genomic sequences render the genomic data inconsistent and uncertain.

Uncertainty in sequential data can be characterised using various representations. One representation accommodating uncertainty is a ***degenerate*** (or ***indeterminate***) ***string*** which manifests when the information about the exact letter at a

given position is not known, but is suspected to be one of the specified letters; this model was first used in the form of "generalised pattern matching" in [Abr87]. A degenerate string is defined by the existence of one or more positions such that each is represented by a set of letters from the alphabet. For instance, $[\begin{smallmatrix}\mathtt{a}\\\mathtt{b}\end{smallmatrix}]\mathtt{ac}[\begin{smallmatrix}\mathtt{b}\\\mathtt{c}\end{smallmatrix}]\mathtt{a}[\begin{smallmatrix}\mathtt{a}\\\mathtt{b}\\\mathtt{c}\end{smallmatrix}]$is a degenerate string over $\Sigma = \{\mathtt{a},\mathtt{b},\mathtt{c}\}$ whereas `abccbaba` is a standard string over the same alphabet.

A specific case of a degenerate string is a representation called a ***partial word*** (introduced in [FP74]) in which every position contains either one letter or the set consisting of all the letters in the alphabet. Usually, an asterisk $*$ or a diamond $\diamond$ represents a *wildcard* symbol (also called a *don't care* symbol or a *hole*) that matches any symbol in the alphabet. A partial word, thus, is a sequence of letters of the set $\Sigma \bigcup \{*\}$.

A ***gapped string*** is another way to capture uncertainty: it is an ordered collection of solid strings separated by variable-length *gaps* defined by an ordered collection of intervals (model introduced in [CS04]). Simply, a gapped string $P$ can be represented as follows [RIL+06]: $P = P_1 *^{a_1,b_1} P_2 *^{a_2,b_2} P_3 \cdots *^{a_{\ell-1},b_{\ell-1}} P_\ell$, where $*$ is a wildcard symbol; $\forall i \in [1,\ell]$ each $P_i$ is a string over $\Sigma$; and $\forall i \in [1,\ell-1]$ each pair $(a_i,b_i)$ represents the gap (minimum and maximum number of wildcard symbols, respectively) between two consecutive strings $P_i$ and $P_{i+1}$.

While a degenerate or a gapped string is an effective representation for *character* (or letter) level uncertainty, it is insufficient to encapsulate a *macro* level of uncertainty that arises in data when it becomes necessary or advantageous to organise multiple distinct-but-similar sequences into a single representation. More specifically, in genomics, an important class of problems is to study intra-species genetic variation; state-of-the-art solutions for this class comprise of matching (*mapping*) short strings (called *reads*) to a longer genomic sequence (canonical *reference genome* obtained through assembly). Owing to the high diversity in biologically relevant genomic regions in many organisms, the population level complexities cannot be captured by the linear structure of a reference genome (see [LKM+14, PNEG17]).

Consequently, the recent research trend has been towards using alternative representations of the genomic sequence (serving as a reference) for population-based genome assembly [HPB13, CSS+15, DCI+15, MdOEMI16]. For example, in human genomics, the reference genome has been represented as a single sequence so far but with the availability of a vast collection of human genomes, the so called *reference cohorts* seem more sensible in order to avoid the reference-bias presented

by a single genomic sequence [PNEG17]. Different representations have recently been explored in an attempt to organise human genomic sequences (which are highly similar) in reference cohorts. Each such representation has its own constraints and challenges.

One such challenge concerns the graphical representation of reference cohorts based on de Bruijn graphs [PTW01], where the representation of data elements is organised around strings of $k$ number of bases, or $k$-*mers*. In a *de Bruijn graph* [dB46], each $k-1$ bases long prefix and suffix of the $k$-mers is represented as a vertex and each $k$-mer is represented as a directed edge between its prefix and suffix vertices. One of the major drawbacks suffered by this model is the problem in defining a *coordinate system* which is an innate advantage of the linear structure [PNH14]. To be able to define a locus on a reference cohort, one should be able to establish a mapping between various graphical motifs and elements or sites in genomic sequences. One such motif called a "superbubble" has been proposed to define the concept of a site in genome. This dissertation proposes an optimal algorithm to identify these structures in a given graph.

Another contribution, in the same context of macro-level uncertainty, presented in this dissertation is the introduction and formalisation of a new notion for representing reference cohorts, which we call "elastic-degenerate strings".

## 1.2 Regularity

Multitudinous problems in genome assembly and inference can be reduced to the core task of finding *regularities* in the sequential genomic data. **Regularity** in the context of strings is an umbrella term used to encapsulate a variety of properties related to the repetitive structure of a string. A few of the typical variants of regularities for a string $w$ have been briefly introduced below:

- **Periodicity [Gus97, p. 42]:** $w$ is *periodic* if it can be expressed as the concatenation of several (more than 1) occurrences of a smaller string (substring), say $u$. In other words, $w = u^p$ such that $p > 1$ ($u^p$ represents $p$ concatenated copies of $u$). The length of the smallest such substring is called the *period*. For example, $w = \mathtt{ababab}$ is periodic with $u = \mathtt{ab}$ (as $w = (\mathtt{ab})^3$).

- **Repeat [Gus97, p. 143]:** A substring $u$ of $w$ is a *repeat* if it has more than 1 occurrences in $w$. The occurrences can be overlapping, adjacent, or non

adjacent. As an illustration, consider $w$ = $\underline{\overline{\texttt{bab}}}\texttt{abbb}\underline{\texttt{bab}}\texttt{a}$. Here, $u$ = bab is a repeat with three occurrences (shown with two underlines and an overline).

- **Repetition [Cro81]:** A substring $u$ of $w$ is a *repetition* if it can be decomposed into two or more adjacent occurrences of a substring $v$ smaller than $u$ i.e. $u = v^p$ (where $p$ in a positive integer greater than 1) and there is no occurrence of $v$ preceding or succeeding $u$. If $p = 2$, $u$ is called a *square*. In $w$ = ba$\underline{\texttt{ababab}}$b, $u$ = ababab (underlined) exemplifies a repetition with $v$ = ab and $p = 3$.

- **Run [Smy02] (or maximal periodicity [Mai89]):** A run is a generalised repetition: a substring $u$ of $w$ is a *run* if it is made up of one or more consecutive copies of $v$ followed by a non-empty prefix of $v$. For example, $v$ = abb constitutes a run (corresponding $u$ is underlined) in $w$ = aa$\underline{\texttt{abbabbabbab}}$a.

- **Cover [AFI91]:** A substring $u$ of $w$ is a *cover* if it has more than 1 overlapping occurrences in $w$ such that each letter of $w$ is in some occurrence of $u$. For instance, $u$ = bab serves as a cover (occurrences demonstrated using underlines and an overline) for $w$ = $\underline{\texttt{bab}}\overline{\texttt{ab}}\underline{\texttt{bab}}$.

- **Seed [IMP96]:** A seed is a generalised cover: a substring $u$ of $w$ is a *seed* if it is a cover such that the first or/and the last occurrences of $u$ are not complete in the sense that the last occurrence can only be a non-empty *prefix* of $u$ and the first occurrence can only be a non-empty *suffix*. For example, $w$ = $\overline{\texttt{ab}}\underline{\texttt{bab}}\overline{\texttt{ab}}\underline{\texttt{bab}}\overline{\texttt{ba}}$ has $u$ = bab as a seed such that its first (virtual) occurrence is a suffix (ab) and the last one is only a prefix (ba).

Detailed surveys of various regularities of strings and their approximate generalisations have been done in [Smy13] and [ZGI08], respectively.

The ability to identify and compute various repeated structures in given strings is known to play a crucial role in many aspects of genomics. Subsequences of DNA that actually code for proteins are interspersed by the non-coding subsequences. In fact, in Eukaryotes, a very small proportion ($< 2\%$ [HL09]) of the genomic sequence accounts for the coding sequences. On the other hand, the non-coding parts of DNA, highly repetitive in structure, have been associated with multiple functions essential for genome functioning [SvS05]. For instance, non coding *regulatory regions* control when and where the expression (synthesis of genetic products like protein) of the genes in their vicinity occur. Moreover, focussing on human genomics, over half

of the genomic sequence has not yet been understood and has been shown to be comprised of repetitive and repeat-derived sequences [dKGC+11].

Usually, occurrences of different forms of regularity are often flanked by regions of interest – genes, for example – which are, in comparison, not regular. In other words, local regularity in a segment of genomic data is indicative of potential *biologically-important regions* for genome-analysis. One of the multiple possible ways to express this notion of local regularity of strings can be in terms of "unbordered factors" of a string. A **border** is one of the central properties characterising regularity associated with the repetitions in a string. A border of a string $w$ is a (possibly empty) *proper factor* of $w$ occurring both as a prefix and as a suffix of $w$. For example, $\varepsilon$ (empty string), a, aa, and aabaa are the borders of $w$ = aabaabaa. A **maximal unbordered factor** is the longest factor of $w$ which does not have a border, e.g. the maximal unbordered factor is aabab for the word $w$ = baabab. With the motivation of capturing the local regular structures in genomic sequences, this dissertation also presents the characterisation of a sequence in terms of its maximal unbordered factors.

Note that the term regularity here is different in meaning from the same term that is generally used to characterise the exons (protein-coding sequences) and introns (non-coding sequences) of DNA (and corresponding RNA transcripts) (see [WPL+16] for example).

## 1.3 Contribution

Substantial work has been done in developing efficient algorithms for processing and analysing biological data. The research corpus, however, still needs significant enrichment, relatively speaking, when it comes to addressing the issue of macro-level uncertainty incorporated therein. Furthermore, due to the computing cost involved and the urgency with which data must be processed and analysed to keep up with the massive incoming volume, efficiency of the tools is imperative. The broader contribution of the research work presented in this dissertation is the development of an assortment of efficient algorithms (and corresponding software tools) to address the uncertainty arising in the representation of an ensemble of sequences and to characterise local regularity present in a sequence in terms of maximal unbordered factors.

This dissertation comprises of a series of algorithms (based on string-specific

algorithms, data-structures, and properties) to solve various important research problems (described below) that find direct or indirect applications in genomic data analysis.

### 1.3.1 Identifying Superbubbles

***Superbubbles*** (a special type of self-contained subgraphs, each with a single source and single sink) are created when a graphical model is used to encode a set of genomes (with variations) as a reference cohort. This structure provides an expressive definition of a site to define a locus in the reference-representation. Identifying these motifs in a reference graph is crucial in order to overcome the limitation of lacking a coordinate system in the graphical representation of reference genomes. We developed an optimal linear algorithm – in terms of both time and space – to identify superbubbles in de Bruijn sequence graphs for genome assembly which is an improvement on the previously best algorithm that runs in $\mathcal{O}(m \log m)$ time, where $m$ is the number of edges in the graph. [*Publication: [BIK+16]*]

### 1.3.2 Pattern-matching in Elastic-degenerate Strings:

We introduced another representation to encapsulate the macro-level uncertainty in sequential data—which we call ***elastic-degenerate strings***—by extending and combining the ideas of gapped strings and degenerate strings. An *elastic-degenerate string* is a string in which an *elastic-degenerate symbol* can occur at one or more positions; each such symbol corresponds to a set of two or more variable-length strings. Another way to visualise an elastic-degenerate string is to see it as an ordered collection of $k > 1$ strings interleaved by $k - 1$ elastic-degenerate symbols. This generalisation of the concept of *degeneracy* is motivated by the advantages of representing a set of related genomes (with variations) as the reference genome for a population. For instance, consider the following set of strings over $\Sigma = \{\texttt{a},\texttt{b},\texttt{c}\}$: {`bccbcaabcabbb, bcaabcacbabb, bcacacacbabb`}. One of their possible alignments is shown below.

| | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| b | c | | c | b | c | a | a | b | c | a | b | b | b |
| b | c | a | a | b | c | a | | | c | b | a | b | b |
| b | c | a | c | a | c | a | c | b | | a | | b | b |

The elastic -degenerate string $\texttt{bc}\begin{bmatrix}\texttt{cb}\\ \texttt{aab}\\ \texttt{aca}\end{bmatrix}\texttt{ca}\begin{bmatrix}\texttt{abcab}\\ \texttt{cba}\end{bmatrix}\texttt{bb}$ is a condensed representation of this set.

We not only formalised the concept of elastic-degenerate strings but also presented a practically efficient algorithm to solve the pattern matching problem in a given elastic-degenerate text. [*Publication: [IKP17]*]

### 1.3.3 Computing Longest Unbordered Factor Array:

As mentioned earlier, a *border* $u$ of a string $w$ is a proper factor of $w$ occurring both as a prefix and as a suffix and the *maximal unbordered factor* of $w$ is the longest factor of $w$ which does not have a border. We developed a quasilinear time ($\mathcal{O}(n \log n)$-time with high probability or $\mathcal{O}(n \log n \log^2 \log n)$-time deterministic) algorithm to compute the *Longest Unbordered Factor Array* of $w$ for general alphabets, where $n$ is the length of $w$. This array specifies the length of the maximal unbordered factor starting at each position of $w$. This is a major improvement on the running time of the currently best worst-case algorithm working in $\mathcal{O}(n^{1.5})$ time for integer alphabets [Gawrychowski et al., 2015]. Moreover, we showed that the analysis of our algorithm is tight: an infinite family of words that exhibit the worst-case behaviour of the algorithm has been provided. [*Publication: [KKMP18]*]

Moreover, for each of the algorithms developed, a software implementation has been done (in C/C++) and made freely available for public dissemination (https://github.com/Ritu-Kundu).

## Author's Contribution:

For each of the algorithms presented in this dissertation, the author has contributed significantly to the formulation of the main idea on which the algorithm is based; the author is the main contributor in the development of the technical details required to transform the main idea into the complete solution as well as in writing up the algorithm for the respective scientific publication; the implementation of the algorithm in the corresponding tool has been done by the author solely.

## 1.4 Outline

The rest of the dissertation is organised in the following format: In Chapter 2, we introduce the fundamental vocabulary, notions, notations, algorithmic tools and data-structures etc. related to strings and graphs along with presenting the basic concepts of genomics and genome sequencing that will be used throughout. However, topic-specific preliminaries have been described only in the respective chapters. Chapter 2 is followed by chapters dedicated to superbubbles (Chapter 3), elastic-degenerate strings (Chapter 4), and the longest unbordered factor array (Chapter 5), respectively. Finally, Chapter 6 concludes the dissertation by summarising the contributions presented in this dissertation and discussing the related open problems and future research directions.

CHAPTER 2

# PRELIMINARIES

Here we present the terminology and basic concepts that are used in the context of String (Sections 2.1, 2.2, and 2.3) and Graph (Section 2.4) Algorithms in order to lay the groundwork for the remainder of this dissertation. The chapter-specific definitions, notations, and data structures have been provided within each chapter. Moreover, in Section 2.5, we present simple biological concepts related to genomics and genome sequencing. We end the chapter with Section 2.6 providing some notational and other conventions used in this dissertation.

## 2.1 Basic Notions and Notations

We begin with basic string-specific definitions and notations.

An **alphabet** $\Sigma$ is a non-empty finite set whose elements are called **letters** (or **characters**); the cardinality of the alphabet set $|\Sigma|$ is called its **size** and is usually denoted with the symbol $\sigma$. An alphabet can be **ordered** (i.e. it has a total ordering of letters) or **unordered** (usually referred to as **general**). An **integer** alphabet is an ordered alphabet where letters are integers from 1 to $\sigma$. In this dissertation, unless stated otherwise, we will consider $\Sigma$ to be ordered and of constant size (i.e. $\sigma = \mathcal{O}(1)$). For instance, the alphabet used for DNA sequences is $\Sigma = \{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}\}$ where $\sigma = 4$.

A **string** (or **word**) is a finite sequence of letters drawn from a fixed alphabet $\Sigma$. The **length** of a string $x$ is denoted by $|x|$. The **empty** string is denoted by $\varepsilon$. For

example, AACGACT is a string of length 7 over the DNA alphabet. A string $x$ of length $n$ can be denoted using either of the following two notations:

- **Sequence Notation:** $x = a_1a_2a_3..a_n$ such that $a_i \in \Sigma \ \forall \ i, 1 \leq i \leq n$.
- **Array Notation:** $x = x[1..n] = x[1]x[2]x[3]..x[n]$ such that $x[i] \in \Sigma \ \forall \ i, 1 \leq i \leq n$; each $i$ is called a **position** or an **index**.

Let $\Sigma^k$ be the set of all finite strings of length $k$ over $\Sigma$; $\Sigma^*$ is the set of all finite strings over $\Sigma$ including the empty string ($\Sigma^* = \Sigma^0 \bigcup \Sigma^1 \bigcup \Sigma^2 ..$); $\Sigma^+$ is the set of all finite strings over $\Sigma$ excluding the empty string ($\Sigma^+ = \Sigma^1 \bigcup \Sigma^2 ..$). Note that $\Sigma^*$ and $\Sigma^+$ themselves are infinite.

The **concatenation** of two strings $u$ and $v$ is the string composed of the letters of $u$ followed by the letters of $v$. It is denoted by $uv$ or also by $u \cdot v$ to show the decomposition of the resulting string. The concatenation operation is associative (i.e. $(uy)v = u(yv)$) but not commutative (i.e. $uv \neq vu$). The empty string $\varepsilon$ is the identity element for the concatenation operation (i.e. $x = \varepsilon x = x\varepsilon$). A string composed of concatenation of $k$ copies of another string u is represented by $u^k$; when $k = 2$, the resulting string is called a **square** (i.e. $x = u^2$ is a square). $x =$ ACAC is an example of a square where $u =$ AC.

For a string $x = x[1..n]$ over $\Sigma$ such that $x = uyv$ where $u, y, v \in \Sigma^*$, the following definitions hold:

- $y$ is a **factor** or **substring**[i] of $x$. If $y \neq x$ then $y$ is a **proper factor** of $x$; $y$ is **non-trivial factor** if it is not empty. In array notation, a non-trivial factor starting at some position $i$ and ending at some position $j$ (i.e. $x[i]x[i+1]..x[j]$, where $1 \leq i \leq j \leq n$) is represented as $x[i..j]$. In this dissertation, we mean a non-trivial factor when we refer to a factor (substring).
- $u$ is a **prefix** of $x$. If $u \neq x$ then $u$ is a **proper prefix** of $x$; $u$ is **non-trivial prefix** if it is not empty. In other words, a non-trivial prefix is a factor starting at position 1 (i.e. $x[1..j]$).

[i] A substring is different from a subsequence because a substring is a contiguous chunk whereas a subsequence of a sequence is the resulting sequence obtained after deleting one or more letters in a possibly non-contiguous fashion.

- $v$ is a **suffix** of $x$. If $v \neq x$ then $v$ is a **proper suffix** of $x$; $v$ is a **non-trivial suffix** if it is not empty. In other words, a non-trivial suffix is a factor ending at position $n$ (i.e. $x[i..n]$).

A string $x$ is **periodic** if it can be expressed as $y^k y'$ where $y \in \Sigma^+$, $k \geq 1$, and $y'$ is a non-trivial prefix of $y$. From another perspective, $x$ is periodic if it is a prefix of $y^{k+1}$ with length $> k|y|$. Here, the length of $y$ is called a *period*. Formally, an integer $p$, $1 \leq p \leq n$, is a **period** of a string $x$ if and only if $x[i] = x[i+p]$ for all $i$, $1 \leq i \leq n-p$. Note that $n$ is always a period of $w$. The smallest period of $x$ is called the **minimum period** (or **the period**) of $x$. If a string is not periodic, it is called **primitive**. For instance, $x =$ `ACACA` is a periodic string with periods 5, 4, and 2 (2 is the period) while the string $x =$ `ACGCC` is a primitive string.

A string $u$ is a **border** of a string $x$, if it is a proper prefix as well as a proper suffix of $x$, i.e. $x = uv = v'u$ for some non-empty strings $v$ and $v'$. Note that the empty word $\varepsilon$ is a border of any word $x$. The longest border ($\neq w$) is referred to as **the border**. For example, `A` and `ACA` are borders of `ACACA` while `ACA` is the border. Period and border are dual of each other and their relationship has been described in more detail in Chapter 5. For a string $x$, a **border array** (or **border table**) is an array $\mathsf{B}$ that records the length of the longest border for each prefix of a string i.e. $\mathsf{B}[i] = \ell$ where $\ell$ is the length of the border of $x[1..i]$. Formally,

$$\mathsf{B}[i] = \begin{cases} \max\{\ell \mid x[1..\ell] = x[i-\ell+1..i]\}, & \text{for } 1 \leq \ell < i, \\ 0 & \text{otherwise.} \end{cases}$$

**Example 2.1.** *Let* $x =$ `aabbabaab`. *The border array is as follows.*

| $i$ | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| $x[i]$ | a | a | b | b | a | b | a | a | b |
| $\mathsf{B}[i]$ | 0 | 1 | 0 | 0 | 1 | 0 | 1 | 2 | 3 |

## 2.2 The Pattern Matching Problem

Two strings $x$ and $y$ over an alphabet $\Sigma$ are said to **match** (represented as $x = y$) if they have equal lengths (say $n$) and each letter of $x$ is the same as that of $y$ at any given position (i.e. $x[i] = y[i]$ $\forall i, 1 \leq i \leq n$). If the corresponding letters at some position are not the same (i.e. $x[i] \neq y[i]$ for some $i, 1 \leq i \leq n$), we say that there is a **mismatch** at that position. Two matching strings $x$ and $y$ are said to be *equal*

in **lexicographic** order; $x$ is lexicographically *smaller* than $y$, denoted as $x < y$, if either $x$ is a proper prefix of $y$ or the letters at the first mismatch position (say $i$) are such that $x[i] < y[i]$.

A shorter string $y$ is said to **occur** (or have an **occurrence**) at some position $i$ in a string $x$ if the substring of $x$ starting at $i$ and with its length equal to $|y|$ matches $y$ (i.e. $x[i\,..\,i+|y|-1] = y$). In the literature, the shorter and longer strings are referred to as the **pattern** and the **text**, respectively; $m$ and $n$ usually denote their respective lengths ($m \leq n$).

The pattern matching problem, arising in numerous applications [Gus97], is to find (or search) all the occurrences, if any, of a given pattern in a given text. More specifically, this is *exact* pattern matching whereas an *approximate* version allows *errors* (consisting of mismatches, insertions, deletions etc.) in the matches. Furthermore, in the exact pattern matching problem, there are many variants:

- when the pattern and the text are given at the time of querying.
- when only the pattern is known beforehand.
- when only the text is known beforehand.

If we know the pattern (or text) in advance, we can pre-process it to answer a search-query faster. Given a pattern $P$ and a text $T$, a search-query itself can take various forms, such as:

- Does $P$ occur in $T$?
- How many occurrences of $P$ are there in $T$?
- What are the positions of the occurrences of $P$ in $T$?

In the classical sense, the pattern matching problem is to report all the occurrences of $P$ in $T$.

A naïve algorithm for searching the matches of a pattern $P$ of length $m$ in a text $T$ of length $n$ is to test a position (say $i$) of $T$ by aligning the beginning of $P$ with $i$ and comparing $P$ and $T$ letter by letter from left to right until either a mismatch is found (implying that there is no occurrence at $i$) or the pattern is exhausted (which implies that $P$ occurs at $i$). We test every position starting from 1 to $n-m+1$ (the last position where the right ends of $T$ and $P$ can be aligned). This process can be visualised as the pattern being *slid* over the text; testing some position and *shifting*

the pattern by one position after every test. The running time of this approach is $\mathcal{O}(nm)$. Below, we present the best known linear-time algorithm for the exact pattern matching problem.

### 2.2.1 The KMP Algorithm and Failure Function

Knuth, Morris, and Pratt (KMP) introduced a linear-time algorithm in [KMP77] for finding all occurrences of a pattern $P$ in a text $T$. The KMP algorithm follows the naïve approach for this problem, that is, it slides $P$ across $T$, albeit shifting here skips the maximum possible number of positions ensuring that no occurrence exists at the skipped positions. The algorithm pre-processes $P$ by computing a **failure function** $f$ that indicates the maximum possible shift using previously performed symbol comparisons. Specifically, the failure function $f(i)$ is defined as the length of the longest prefix of $P$ that is a suffix of $P[1..i]$ [ii]; in fact, the failure function is the border array of $P$. Note that the failure function can be constructed in time linear in the length of the string i.e. in $\mathcal{O}(m)$ time for $P$.

The failure function is used as follows while searching: suppose a position $i$ of $T$ is being tested and there is a mismatch after $k$ letters (i.e. $T[i..i+k-1] = P[1..k]$ but $T[i+k] \neq P[k+1]$). If $f(k) = \ell$ (i.e. $P[1..\ell] = P[k-\ell+1..k]$), then the shift is $k-\ell$ due to the associativity of the matches. Similarly, if an occurrence has been found at some position, the shift will be $m - f(m)$. Consequently, by using the failure function, the algorithm achieves an optimal search time of $\mathcal{O}(n)$ after $\mathcal{O}(m)$-time pre-processing.

## 2.3 Fundamental Data Structures

In the following, we present three prominent data structures supporting a wide variety of string matching algorithms. In particular to this dissertation, these data structures mainly serve the purpose of answering the **Longest Common Prefix** (**LCP**) queries, defined as follows: "Given two indices $i$ and $j$ of a string, what is the longest prefix common to both the suffixes that start at positions $i$ and $j$?". Usually an LCP query is denoted by a function call LCP$(i,j)$. For example, LCP$(2,5)$ on

[ii] An optimised version of KMP algorithm uses an additional condition, namely, if $f(i) = \ell$ then the letters $P[\ell+1]$ and $P[i+1]$ are different.

$x$ = aabbabbaa is abba because it is the longest prefix of the suffixes starting at positions 2 (abbabbaa) and 5 (abbaa); whereas LCP(3,5) is $\varepsilon$.

### 2.3.1 Suffix Tree:

The **suffix tree** $\mathscr{S}(x)$ of a non-empty string $x$ of length $n$ over a fixed-sized alphabet is a compact trie representing all the suffixes of $x$ such that $\mathscr{S}(x)$ has $n$ leaves labelled from 1 to $n$. Each internal node, other than the root, has at least two children and each edge is labelled with a non-empty factor of $x$. No two edges out of a node can have edge-labels beginning with the same letter. If v is a node of $\mathscr{S}(x)$, then the *path-label* of v is the concatenation of the edge labels along the path from the root to v; the length of the path-label is the *string-depth* of node v. For any $i$, $1 \leq i \leq n$, the path-label of the terminal node $i$ is precisely the suffix $x[i..n]$. Note that, if the last letter of $x$ is unique then every terminal node is a leaf i.e. every suffix ends in a leaf node. In order to have a one-to-one correspondence between the leaf nodes and the suffixes, we usually append a unique symbol (typically a "$" such that $\$ \notin \Sigma$) to $x$.

Additionally, for any two suffixes $u = x[i..n]$ and $v = x[j..n]$ of $x$, if $w$ is the LCP of u and v, then the path in $\mathscr{S}(x)$ corresponding to $w$ is the same for u and v. In other words, the string-depth of the **Lowest Common Ancestor** (LCA) node of the two leaves is the same as the length of the LCP of the suffixes represented by those leaves. For a general introduction to suffix trees, see [CHL07].

The construction of the suffix tree $\mathscr{S}(x)$ takes $\mathscr{O}(n)$ time and space using one of the several seminal algorithms: Weiner's [Wei73], McCreight's [McC76], or Ukkonen's [Ukk95]. Once the suffix tree of $x$ has been constructed, the LCA of any two leaves of $\mathscr{S}(X)$, and thus the length of the LCP of any two suffixes of $x$, can be computed in constant time after a linear-time pre-processing [HT84, SV88]. In addition, it can be used to support queries that return all the occurrences of a given pattern of length $m$ in time $\mathscr{O}(m+z)$ where $z$ is the number of occurrences.

A **generalised suffix tree** is a suffix tree constructed for a set of strings [AFG+94, Gus97]. It can be obtained, for a given set of $l$ strings $\{x_1, x_2, \cdots, x_l\}$ over $\Sigma$ with total combined length $N$ (i.e. $\sum_{i=1}^{l} |x_i| = N$), by constructing the suffix tree of the concatenated string $x_1\$_1x_2\$_2\cdots x_l\$_l$, where each $\$_i$ $\forall i \in [1\cdots l]$ is unique end-marker for each string such that $\$_i \notin \Sigma$ and $\$_i \neq \$_j$ $\forall i, j \in [1\cdots l]$. It should be clear that the construction of the generalised suffix tree requires $\mathscr{O}(N)$ time.

### 2.3.2 Enhanced Suffix Arrays

We denote by SA the **suffix array** [MM93] of a string $x$ of length $n$. SA is an integer array of size $n$ storing the starting positions of all the (lexicographically) sorted non-empty suffixes of $w$, i.e. for all $2 \leq r \leq n$ we have $x[\mathsf{SA}[r-1]..n] < x[\mathsf{SA}[r]..n]$; $\mathsf{SA}[r] = i$ implies that suffix starting at $i$ has rank $r$ in the sorted order. Effectively, SA keeps the leaf order of the suffix tree (with edges ordered lexicographically based on their labels) of $x$. SA, together with other auxiliary arrays, is known as the **enhanced suffix array** [AKO02]; one type of the auxiliary arrays keeps the lengths of the LCPs of lexicographically consecutive suffixes (i.e. the position $i$ in this array stores the length of the LCP of suffixes that have ranks $i$ and $i-1$). An enhanced suffix array can answer LCP queries in constant time and can be constructed in $\mathcal{O}(n)$ space and $\mathcal{O}(n)$ time for integer alphabets [MM93, BFC00, KLA+01].

### 2.3.3 RMQ

The Range Minimum (or Maximum) Query problem, RMQ for short, is to pre-process a given array $\mathsf{A}[1..n]$ for subsequent queries of the form: "Given indices $i, j$, what is the minimum (or maximum) value of $\mathsf{A}[i..j]$?". The problem has been studied intensively for decades and several $\langle\mathcal{O}(n),\mathcal{O}(1)\rangle$-RMQ data structures (i.e. linear-time pre-processing and constant-time to answer queries) have been proposed, many of which depend on the equivalence between the Range Minimum (or Maximum) Query and the Lowest Common Ancestor problems [HT84, FH06, Dur13].

## 2.4 Fundamentals of Graphs

A graph is a model to represent relationships between various entities (denoted by nodes) using arcs. In this section, we present some fundamental notions, definitions, and techniques related to graphs which will be used in the later chapters. Further details of the presented concepts can be found in [THCS01].

Formally, a **graph** $G = (\mathbb{V}, \mathbb{E})$ consists of a set $\mathbb{V}$ of **vertices** (nodes) and a set $\mathbb{E}$ of **edges** (arcs). An edge in $\mathbb{E}$ between a vertex $\mathtt{u}$ and a vertex $\mathtt{v}$ ($\mathtt{u}, \mathtt{v} \in \mathbb{V}$) is denoted as the pair $(\mathtt{u}, \mathtt{v})$. Typically, the number of vertices and edges in a graph are represented by positive integers $n$ and $m$, respectively ($|\mathbb{V}| = n, |\mathbb{E}| = m$).

A graph is said to be **undirected** if an unordered pair represents an edge (i.e. an edge $(\mathtt{u}, \mathtt{v})$ is same as the edge $(\mathtt{v}, \mathtt{u})$); otherwise the graph is **directed**. Some of

the common definitions in the context of graphs are as follows:

- Vertices $\mathtt{u}$ and $\mathtt{v}$ are **adjacent vertices** if and only if $(u, v)$ is an edge in the graph. The edge $(\mathtt{u}, \mathtt{v})$ in an undirected graph is said to be **incident on** vertices $\mathtt{u}$ and $\mathtt{v}$ whereas in a directed graph it is referred to as incident *from* $\mathtt{u}$ *to* $\mathtt{v}$.

- For an undirected graph, the **degree** $d_{\mathtt{u}}$ of a vertex $\mathtt{u}$ is the number of edges incident on $\mathtt{u}$. The analogous concept for a directed graph is that of **in-degree** and **out-degree**; the in-degree $d^{in}_{\mathtt{u}}$ of a vertex $\mathtt{u}$ is the number of edges incident to $\mathtt{u}$ (*incoming edges*); the out-degree $d^{out}_{\mathtt{u}}$ of a vertex $\mathtt{u}$ is the number of edges incident from $\mathtt{u}$ (*outgoing edges*). A vertex having zero in-degree (i.e. no incoming edges) is said to be a **source** vertex of the graph. Similarly, a vertex with zero out-degree (no outgoing edges) is referred to as a **sink** vertex. The vertices connected to some vertex in an undirected graph or via outgoing edges in a directed graph are called the **neighbours** of that vertex.

- A **path** $P$ from vertex $\mathtt{v}_1$ to vertex $\mathtt{v}_k$ is a sequence of vertices $P =< \mathtt{v}_1, \mathtt{v}_2, .., \mathtt{v}_k >$ such that $(v_i, v_{i+1}) \in \mathbb{E}\ \forall i,\ 1 \leq i \leq k$. $P$ is said to be **simple** iff the vertices are unique. A vertex $\mathtt{v}$ is said to be **reachable** from another vertex $\mathtt{u}$ if there is a path from $\mathtt{u}$ to $\mathtt{v}$. A **cycle** is a path such that $\mathtt{v}_1 = \mathtt{v}_k$. A cycle is said to be *simple* if its vertices (except the first and the last) are unique.

- An undirected graph is **connected** if every vertex is reachable from every other vertex. Analogously, a directed graph is said to be **strongly connected** if every ordered pair of vertices is connected via a path.

- A **Directed Acyclic Graph** (**DAG**) is a directed graph without cycles. In a DAG, we refer to a vertex $\mathtt{v}$ connected to a vertex $\mathtt{u}$ such that $(\mathtt{v}, \mathtt{u}) \in \mathbb{E}$ as a **parent** of $\mathtt{u}$; $\mathtt{u}$ is called a **child** of $\mathtt{v}$.

- An acyclic connected graph is called a **tree** whereas an acyclic possibly disconnected graph is called a **forest**. If some vertex of a tree has been labelled as the **root**, it becomes a **rooted tree**. Vertices of a rooted tree are usually referred to as **nodes**. Note that there is a single unique path between any two nodes of a tree. A rooted tree can be made directed by giving it an orientation – every edge points away from the root (called an **arborescence**) or towards it (called an **anti-arborescence**). In a rooted tree, if a node $\mathtt{u}$ is on the path from the root to some node $\mathtt{v}$ then $\mathtt{u}$ is called an **ancestor** of $\mathtt{v}$ and $\mathtt{v}$ is called

a **descendant** of u; if u is the last node on this path then u is referred to as the **parent** of v and v is called a **child** of u.

- A graph $G' = (\mathbb{V}', \mathbb{E}')$ is a **subgraph** of another graph $G = (\mathbb{V}, \mathbb{E})$ if $\mathbb{V}' \subset \mathbb{V}$, $\mathbb{E}' \subset \mathbb{E}$, and an edge $(u, v) \in \mathbb{E}'$ implies that $u, v \in \mathbb{V}'$. In other words, a subgraph contains a subset of the vertices of the original graph and a subset of the edges between only those vertices. If the subgraph contains *all* the edges (present in the original graph) between the vertices *selected* by the subgraph then it is called an **induced subgraph** (more specifically, a *vertex induced* subgraph). In this dissertation, we will refer to a vertex induced subgraph as simply a subgraph.

### 2.4.1 Depth First Search

Visiting (thereby processing) every vertex of a graph while keeping the *redundancy* (visiting the same vertex again) minimum is called the **graph traversal problem**. One of the strategies for traversing a given graph is **Depth First Search** (DFS) which proceeds by going as *deep* in the graph as possible and *backtracking* (going back) when it encounters a dead-end. The DFS algorithm visits an unexplored vertex (say v) to begin, then visits one of its unexplored adjacent vertices (say w), then moves on to exploring vertices adjacent to w, and so on; if w has no unexplored neighbour, it backtracks to v and starts exploring the other unexplored neighbours of v. We usually label vertices as 'unvisited' , 'finished', and 'discovered' to indicate their current states – an unexplored vertex is labelled 'unvisited'; a vertex which has no neighbour left to explore is labelled 'finished'; 'discovered' is used for a vertex which has been visited but has some neighbours which have not yet been marked 'finished'. Initially all vertices are labelled 'unvisited'. The algorithm stops when the initial vertex is labelled 'finished'. Note that if the graph is not connected, DFS will be repeated starting from some other unexplored vertex until all the vertices have been explored. DFS algorithm uses a stack explicitly or implicitly (via recursion) to realise this particular order of visits. The running time of DFS is linear in the size of the graph i.e. $\mathcal{O}(n + m)$.

In effect, DFS produces a **spanning tree** of the given graph. A spanning tree of a graph $G$ is a tree that is a subgraph containing all the vertices of $G$. We will refer to the spanning tree resulting from DFS as the **DFS-tree**; the vertex with which DFS begins is the *root*. We obtain a single DFS-tree if the graph is connected, otherwise the result is a DFS-forest of multiple DFS-trees. The edges of a graph that

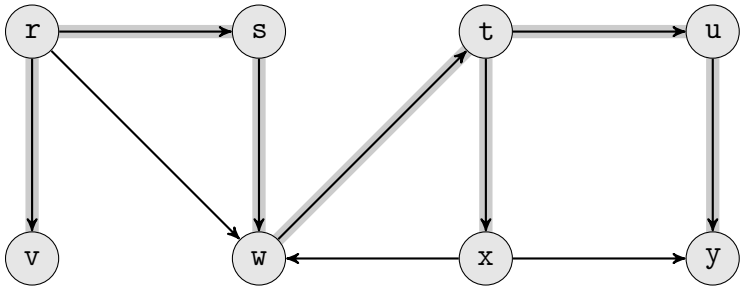


Figure 2.1: An illustration of a DFS-tree. The tree edges have been highlighted. The remaining edges are such that (r, w) is a forward edge, (x, w) is a back edge; (x, y) is a cross edge. Observe the cycle < w, t, x, w >.

constitute its DFS-tree are called **tree edges**. The remaining edges can be classified into the following three categories:

- **Forward edges**: An edge (u, v) is a *forward edge* if v is a *descendent* of u in the DFS-tree.
- **Back edges**: An edge (u, v) is a *back edge* if v is an *ancestor* of u in the DFS-tree. In DFS, v is labelled 'discovered' when the edge (u, v) is checked.
- **Cross edges**: An edge that is neither a forward edge nor a back edge is a *cross edge*.

A cyclic graph will have at least one back edge whereas an acyclic graph contains none. Figure 2.1 illustrates a DFS-tree rooted at vertex r (when the neighbours are selected lexicographically) and the associated classification of the edges.

### 2.4.2 Topological Sort

A **topological sort** of a DAG $G = (\mathbb{V}, \mathbb{E})$ is a linear ordering of all its vertices such that if $G$ contains an edge $(u, v)$, then u appears before v in the ordering. There exists a classical linear-time ($\mathcal{O}(n+m)$) algorithm for computing the topological ordering of a directed acyclic graph [THCS01, Tar76]. In its recursive form, the algorithm visits an unvisited vertex of the graph, finds its unvisited neighbour, say v, and performs another topological sort starting from v. The algorithm *returns* if the current vertex does not have unvisited neighbours. An example of topological sort has been shown in Figure 2.2.

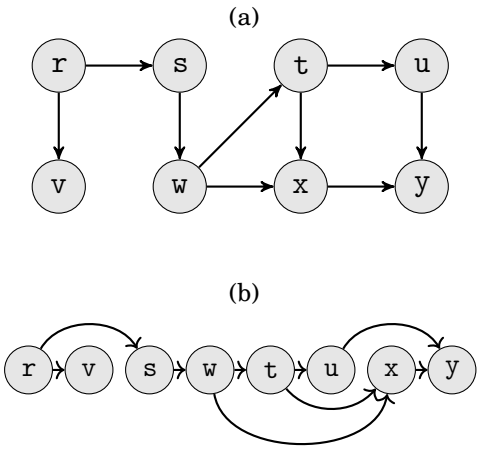


FIGURE 2.2. Vertices of the DAG shown in Figure (2.2(a)) are arranged in topological order in Figure (2.2(b)).

### 2.4.3 Strongly Connected Components

A **strongly connected component** (**SCC**) of a directed graph is a *maximal* subgraph that is strongly connected (i.e. for every pair of vertices $(u, v)$ in this subgraph, there is a path from $u$ to $v$); this subgraph is maximal in the sense that the subgraph resulting from inclusion of any additional vertex in its vertex set will not be strongly connected. An SCC is said to be **singleton** if it contains only one vertex; otherwise it is **non-singleton**. There are several well known algorithms based on depth first search which find the strongly connected components of a given DAG in linear-time ($\mathcal{O}(n+m)$) [Sha81, Tar72, Dij97]. Figure 2.3 demonstrates the strongly connected components in the given graph.

### 2.4.4 De Bruijn Graph

In graph theory, the standard $\ell$*−dimensional* de Bruijn graph $G = (\mathbb{V}, \mathbb{E})$ [dB46] for a given alphabet $\Sigma$ is such that all the strings over $\Sigma$ of length $\ell$ constitute $\mathbb{V}$ and for every pair of vertices with an overlap of length $\ell - 1$, there is an edge in $\mathbb{E}$. More precisely, if the suffix of length $\ell - 1$ of a vertex $u$ matches the prefix of length $\ell - 1$ of another vertex $v$ then $(u, v) \in \mathbb{E}$. Note that $u$ and $v$ need not be distinct (i.e. edges like $(u, u)$ are possible).

In the context of genomics, a modified version of a de Bruijn graph is used which is built for a given set of strings $\mathbb{R}$ and a positive integer $k > 1$. The vertices

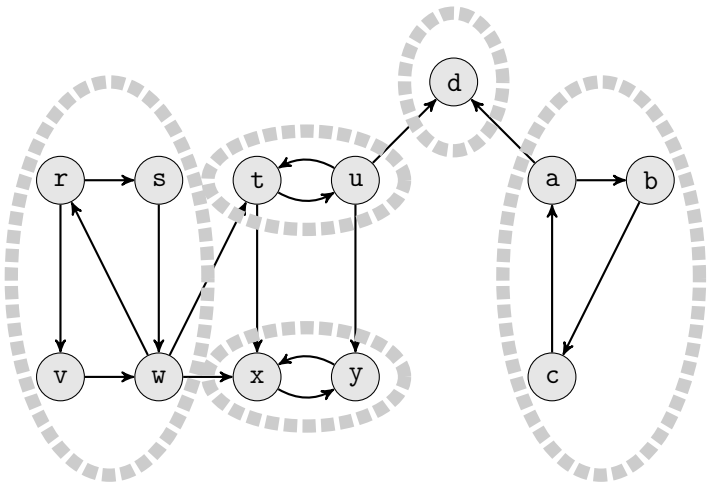


Figure 2.3: There are five strongly connected components in this graph (marked using the dotted elliptical shapes). The SCC consisting of the vertex d is singleton; all others are non-singletons.

in this modified de Bruijn graph consist of all the distinct substrings of length $k$ (called *k-mers*) of the strings in $\mathbb{R}$. The edge set is obtained in the standard[iii] way: for all $u, v \in \mathbb{V}$, an edge $(u, v)$ is added if the $k-1$-length suffix of $u$ matches the $k-1$-length prefix of $v$. Figure 2.4 shows the de Bruijn graph corresponding to $\mathbb{R} = \{CAAAAT, CAATG\}$ and $k = 3$.

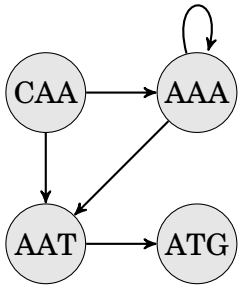


FIGURE 2.4. De Bruijn graph corresponding to $\mathbb{R} = \{CAAAAT, CAATG\}$ and $k = 3$.

[iii]This is a simplified version. In reality, there are weights on the edges reflecting the number of times the $k-length$ substring has appeared in $\mathbb{R}$. In this dissertation, we do not consider the weights on the edges.

## 2.5 Basic Concepts of Genomics

In this section, we present an extremely simplified version of some biological notions and gene sequencing which will help in understanding the biological context of the problems for which algorithms have been presented in this dissertation. We refer the reader to [MBCT15] for a detailed combinatorial perspective of the presented concepts.

### 2.5.1 DNA, RNA, and Protein Sequences

**DNA** (Deoxyribonucleic acid) is a biomolecule carrying the genetic information necessary for reproduction, growth, and functioning of living organisms (and some viruses). It is a chain of building blocks called **nucleotides**; each nucleotide contains one of the four **bases** – cytosine (C), guanine (G), adenine (A) or thymine (T). From an informatics perspective, DNA can be seen as a string over an alphabet $\Sigma = \{A, C, G, T\}$. DNA usually occurs in a **double stranded** form (i.e. two strands or chains) intertwined in a double helical structure. The pairing between bases – A with C and G with T – keeps the double helix stable. As a result, the strands are **complimentary** to each other i.e. one strand can be obtained from the other by simply replacing A with C (and vice versa) and G with T (and vice versa). Physically, DNA is usually present in a condensed form called **chromosomes**, and the complete set of all the DNA sequences of an organism is called a **genome** [iv]. A genome can be as long as a few million base pairs (in bacteria) or more than a hundred billion base pairs (human genome is about 3 billion base pairs long).

**Proteins** are biomolecules responsible for a wide range of essential functions required in a life-form. A protein is a chain of smaller units called **amino acids** folded into complex three-dimensional structures. Most proteins are made up of up to 20 different amino acids. Thus, a protein molecule can be primarily thought of as a string over an alphabet consisting of 20 letters. Protein sequences are **encoded** in subsequences of DNA; such subsequences are called **genes**. Typically, in complex life forms, a gene consists of short substrings called **exons** interspersed by large substrings called **introns**. An ordered subset of exons, called a **transcript**, typically corresponds to one protein. As a result, the same gene can have multiple transcripts and thus encode multiple proteins. Encoding from DNA to protein is usually a three

[iv] In most viruses, genome is composed of RNA (rather than DNA) sequences

step process – *transcription*, *splicing*, and *translation*. In **transcription**, the two strands open up and a complementary (with respect to one of the strands) **RNA** molecule is produced; chemically, RNA is same as DNA with the only difference being that `T` is replaced with another base `U` (uracil). Transcription is followed by **splicing** (cutting off) introns to combine subsets of exons so as to produce one or more transcripts. The result of the splicing process is called *mRNA* (matured messenger RNA). In the final step i.e. **translation**, the mRNA is *read* sequentially from left to right encoding a triplet of bases (called a **codon**) into specific amino acids which are chained together to form the corresponding protein. The *translation table* associating such triplets to amino acids is shared by most of organisms and is called the **genetic code**.

The rate of transcription is controlled (inhibited or enhanced) by the binding of specific proteins called **transcription factors** in specific regions called **regulatory regions**. The DNA substrings to which transcription factors bind are called **transcription factor binding sites** (**TFBS**). These are located in either the **promoter** region (a 100-1000 base pair long region, which initiates the transcription process, situated near the site at which the transcription of a gene starts) or at a large sequential distance from the gene.

### 2.5.2 DNA Sequencing and Variant Calling

The genomes of individuals belonging to the same species typically have the same number of chromosomes and by and large the same base sequences in a chromosome. Consequently, a **consensus** or **reference** genome can represent a typical genome associated with a species. However, **mutations** (permanent alteration of sequence of a gene) and *recombination* (random cross over of chromosomes inherited from *mother* and *father* in *sexual reproduction*) can cause **genetic variations** as the genome is copied from cell to cell or from individual to individual across generations. Variations are usually small scale – mostly consisting of changes in single bases (single nucleotide polymorphism or **SNPs**) and less frequently, insertion or deletions of bases (**InDels**). Every possible variant found at some specific position in a chromosome is called an **allele** (i.e. a different form of the same gene).

DNA sequencing is the process of inferring the base sequence that constitutes a DNA sequence. DNA sequencing can be done for the whole genome (called **whole genome sequencing**) or only specific portions (for example, only exons of genes). To date, sequencing technologies have not advanced to the level where an entire

chromosome can be (accurately) sequenced. The most-employed technique by the state-of-the-art sequencers is to fragment a long sequence into smaller sequences randomly, followed by the creation of copies of each fragment (amplification) and then sequencing each fragment; each such sequenced fragment is called a **read**. In this process, the information about the relative placement of the reads with respect to the DNA is lost. Reads must be overlapping in order to have sufficient information to stitch them together. Therefore, several rounds of this *fragment-amplify-sequence* process are repeated. Stitching the reads together to infer the DNA sequence is called **fragment assembly** or **genome assembly** – a non-trivial combinatorial problem. Various other factors such as errors in reads while sequencing, repeats in the DNA, variations etc. make the assembly problem even more complex.

Assembly can be categorised as **de novo** – when the reference genome is not known – or **mapping or resequencing** otherwise. In mapping assembly, reads are **mapped** or **aligned** to the most *similar* (based on some *similarity* or *distance* measure) fragment in the reference genome. Subsequently, variations in the aligned reads can be identified with respect to the reference (**variant calling**).

## 2.6 Conventions

For the algorithms presented in this dissertation, we assume the *word-RAM* model [FW90] of computation with $\Omega(\log n)$ bits in a computer-word (where $n$ is the length of the string in consideration). Analysis of space (memory) is in terms of computer-words. Other conventions and notations being followed (unless specified otherwise) are as follows

- log is to the base 2.
- We use the term *algorithm* for the pseudo-code specifying the main algorithm solving the problem in consideration and *subroutine* for a module assisting the main algorithm which consists of one or more functions.
- We denote a set using either curly brackets ({}) with its elements separated by commas (,), or square brackets ([]) with the elements stacked vertically. A list is denoted by elements separated by commas encapsulated in square brackets.
- $x, y, u, v$ etc. are used to denote strings; an exception is using $T$ or $P$ for strings representing the text and the pattern respectively in the pattern matching

problem. Other representations have been listed below (examples listed in the left column with the corresponding representations in the right column):

| | |
|---|---|
| $i, j, k, n, m, p$ | integers |
| `u, v, p, c` | vertices (`teletype` font family) |
| `a, b, c, d` | alphabet-letters (`teletypefont` family) |
| ALGONAME | name of algorithms / subroutines / functions |
| $\mathbb{S}$ | sets |
| Array | arrays |
| $\mathit{List}$ | lists |
| $\mathcal{T}$ | trees |

CHAPTER 3

# SUPERBUBBLES

DNA sequencing is the process of determining the exact order of the nucleotide bases in an individual's genome in order to catalogue the sequence variation and understand its biological implications. Whole-genome sequencing techniques produce masses of data in the form of short sequences known as reads. Assembling these reads into a whole genome is a major algorithmic challenge. Most assembly algorithms utilise de Bruijn graphs [dB46] constructed from the reads for this purpose. A critical step of these algorithms is to detect typical motif structures in the graph; one such complex subgraph class is the so-called *superbubble*. In this chapter, we propose an $\mathcal{O}(n+m)$-time algorithm to detect all superbubbles in a directed acyclic graph with $n$ vertices and $m$ (directed) edges, improving the best-known $\mathcal{O}(m \log m)$-time algorithm by Sung et al [SSS+15].

This chapter is organised as follows: we begin by providing the background and reviewing related literature in Section 3.1. In Section 3.2, we define superbubbles, introduce some of their properties, and give an overview of the previous state-of-the-art algorithm. In Section 3.3, we outline the $\mathcal{O}(n+m)$-time algorithm for computing superbubbles in a directed acyclic graph. We describe a method to validate a candidate superbubble in constant time in Section 3.4. The algorithm is analysed in Section 3.5. Finally, in Section 3.6, we brief the reader on the impact of the contribution presented in this chapter.

## 3.1 Background

Since the publication of the first draft of the human genome [LLB+01, VAM+01], the field of genomics has changed dramatically. Recent developments in sequencing technologies (see [Bal11], for example) have made it possible to sequence new genomes at a fraction of the time and cost required only a few years ago. With applications such as sequencing the genome of a new species, an individual within a population, and RNA molecules from a particular sample, sequencing remains at the core of genomics.

Whole-genome sequencing creates masses of data, in the order of tens of gigabytes, in the form of short sequences (reads). Genome assembly involves piecing together these reads to form a set of contiguous sequences (contigs) representing the DNA sequence in the sample. Traditional assembly algorithms rely on the *overlap-layout-consensus* approach [Bat05], representing each read as a vertex in an *overlap graph* and each detected overlap as a directed edge between the vertices corresponding to the overlapping reads. These methods have proved their use through numerous *de novo* genome assemblies [BMK+08]. Please refer to Subsection 2.5.2 for a general introduction to sequencing and assembly processes.

Subsequently, a fundamentally different approach based on de Bruijn graphs was proposed [PTW01], where representation of data elements was organised around the words of $k$ nucleotides, or $k$-mers, instead of reads. Unlike in an overlap graph, in a *de Bruijn graph* (as described in Subsection 2.4.4), each $k-1$ nucleotide long prefix and suffix of the $k$-mers is represented as a vertex and each $k$-mer is represented as a directed edge between its prefix and suffix vertices. The marginal information contained in a $k$-mer is its last nucleotide. In a de Bruijn graph, the assembly problem is (ideally) reduced to finding an Eulerian path, that is, a trail that visits each edge in the graph exactly once.

However, sequencing errors and genome repeats significantly complicate the de Bruijn graph by adding false vertices and edges to it. Efficient and robust filtering methods have been proposed to simplify the graph by filtering out motifs such as *tips*, *bubbles*, and *cross links*, which proved to be caused by sequencing errors [ZB08]. In particular, a bubble consists of multiple directed unipaths where a unipath is a path in which all internal vertices are of degree 2, from a vertex `v` to a vertex `u` and is commonly caused by a small number of errors in the centre of the reads. Although these types of motifs are simple and can easily be identified and filtered out, more

complicated motifs prove to be more challenging.

Recently, a complex generalisation of a bubble, the so-called superbubble, was proposed as an important subgraph class for analysing assembly graphs [OSS13]. A *superbubble* is defined as a minimal subgraph $H$ in the de Bruijn graph with exactly one start vertex s and one end vertex t such that (1) $H$ is a directed, acyclic, single-source (s), single-sink (t) graph (2) there is no edge from a vertex not in $H$ going to a vertex in $H \backslash \{s\}$, and (3) there is no edge from a vertex in $H \backslash \{t\}$ going to a vertex not in $H$. Please note that the definition of superbubbles is general (i.e. not restricted to de Bruijn graphs only); consequently, the algorithms mentioned in this chapter can be applied to any directed graph for finding superbubbles.

Superbubbles – originally associated with sequencing errors, inexact repeats, diploid/polyploid genomes, or frequent mutations [OSS13] – have recently been proposed to be used as definitions of sites (in the context of allele calling) [PNEG17]. Motifs like superbubbles emerge when new variants are added to the graphical model of a reference cohort. Because of its ability to capture the nested relationships between variants, a superbubble can overcome the lack of a coordinate system which is a major limitation of graph-centred modelling of reference cohorts. A general introduction to the concepts of a reference genome, genetic variations, allele calling etc. has been provided in Subsection 2.5.2.

Onodera et al. [OSS13] gave the first algorithm to detect superbubbles that runs in $\mathcal{O}(nm)$ time, where $n$ is the number of vertices and $m$ is the number of edges in the graph. Given a directed graph $G = (V, E)$, this algorithm proceeds by iterating a search step for each vertex with an assumption that it might be the source of a superbubble. A search step visits vertices in the standard topological order, starting from a given vertex s, to eventually report a vertex t such that $\langle \mathtt{s}, \mathtt{t} \rangle$ is a *superbubble* (if any).

Subsequently, Sung et al. [SSS+15] gave an improved $\mathcal{O}(m \log m)$-time algorithm to solve this problem. Their algorithm partitions the given graph into a set of subgraphs such that the set of superbubbles in all these subgraphs is the same as the set of superbubbles in the given graph. Superbubbles are then detected in each subgraph; if it is cyclic, it is first converted into a directed acyclic subgraph by duplicating vertices (and some edges) and employing depth-first search.

**Our Contribution**. The cost of partitioning the graph and transforming it into the directed acyclic subgraphs, in the algorithm by Sung et al., is linear with respect to the size of the graph. However, computing the superbubbles in each directed

acyclic subgraph requires an overall $\mathcal{O}(m \log m)$ time, which dominates the time bound of the algorithm. We propose a new $\mathcal{O}(n+m)$-time algorithm to compute all the superbubbles in a directed acyclic graph which eliminates this bottleneck, resulting in an optimal linear-time algorithm overall.

**Software Tool**. The software tool implementing the presented algorithm as well as the prior stages of generating the directed acyclic subgraphs from a given (general) graph has been developed and made freely available for public dissemination (on Github [i]).

## 3.2 Preliminaries

The concept of superbubbles was introduced and formally defined in [OSS13] as follows.

**Definition 3.1** ([OSS13])**.** Let $G = (\mathbb{V}, \mathbb{E})$ be a directed graph. For any ordered pair of distinct vertices $\mathtt{s}$ and $\mathtt{t}$, $\langle \mathtt{s}, \mathtt{t} \rangle$ is called a ***superbubble*** if it satisfies the following:

- **reachability:** $\mathtt{t}$ is reachable from $\mathtt{s}$;
- **matching:** the set of vertices reachable from $\mathtt{s}$ without passing through $\mathtt{t}$ is equal to the set of vertices from which $\mathtt{t}$ is reachable without passing through $\mathtt{s}$;
- **acyclicity:** the subgraph induced by $\mathbb{U}$ is acyclic, where $\mathbb{U}$ is the set of vertices satisfying the matching criterion;
- **minimality:** no vertex in $\mathbb{U}$ other than $\mathtt{t}$ forms a pair with $\mathtt{s}$ that satisfies the conditions above;

vertices $\mathtt{s}$ and $\mathtt{t}$, and $\mathbb{U} \backslash \{s, t\}$ used in the above definition are the superbubble's ***entrance***, ***exit*** and ***interior***, respectively.

We note that a superbubble $\langle \mathtt{s}, \mathtt{t} \rangle$ in the above definition is equivalent to a single-source, single-sink, directed acyclic subgraph of $G$ with source $\mathtt{s}$ and sink $\mathtt{t}$, which does not have any *cut vertices* (a cut vertex or articulation point in a connected graph is the vertex which when removed from the graph along with the edges associated

[i] https://github.com/Ritu-Kundu/Superbubbles

with this vertex, results in a disconnected graph) and preserves all in-degrees and out-degrees of vertices in $\mathbb{U}\backslash\{\mathtt{s},\mathtt{t}\}$, as well as the out-degree of $\mathtt{s}$ and in-degree of $\mathtt{t}$.

Formally, the problem of identifying the superbubbles in a directed acyclic graph $G$ can be defined as follows:

| IDENTIFICATION OF SUPERBUBBLES |
| --- |
| **Input:** A directed acyclic graph $G = (\mathbb{V}, \mathbb{E})$. |
| **Output:** All the superbubbles $\langle \mathtt{s}, \mathtt{t} \rangle$ where $\mathtt{s}$ and $\mathtt{t}$ are in $\mathbb{V}$. |

As an illustration, consider a directed graph as shown in Figure 3.1. There are five superbubbles in this graph: $\langle \mathtt{v}_1, \mathtt{v}_3 \rangle$, $\langle \mathtt{v}_3, \mathtt{v}_8 \rangle$, $\langle \mathtt{v}_5, \mathtt{v}_7 \rangle$, $\langle \mathtt{v}_{11}, \mathtt{v}_{12} \rangle$ and $\langle \mathtt{v}_8, \mathtt{v}_{14} \rangle$. Here, both $\langle \mathtt{v}_5, \mathtt{v}_7 \rangle$ and $\langle \mathtt{v}_{11}, \mathtt{v}_{12} \rangle$ are nested superbubbles.

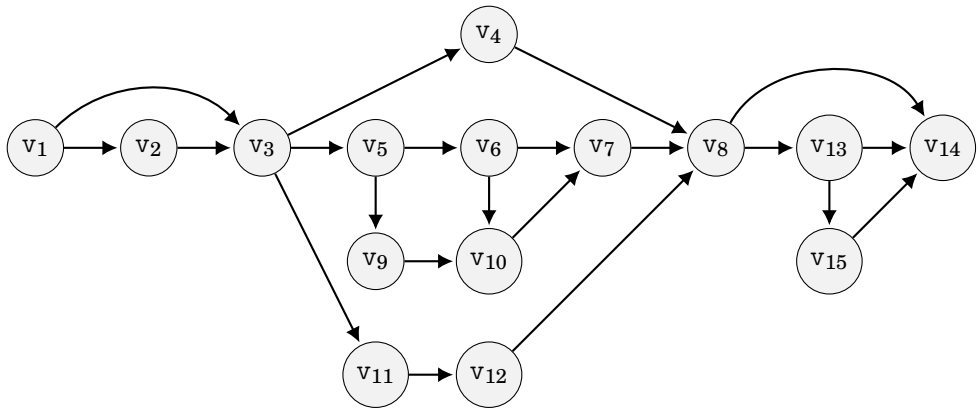


Figure 3.1: A graph $G$ with set of vertices $\mathbb{V} = \{\mathtt{v}_1, \mathtt{v}_2, \cdots, \mathtt{v}_{15}\}$ and five superbubbles: $\langle \mathtt{v}_1, \mathtt{v}_3 \rangle$, $\langle \mathtt{v}_3, \mathtt{v}_8 \rangle$, $\langle \mathtt{v}_5, \mathtt{v}_7 \rangle$, $\langle \mathtt{v}_{11}, \mathtt{v}_{12} \rangle$ and $\langle \mathtt{v}_8, \mathtt{v}_{14} \rangle$.

We next state a few important properties of superbubbles which enable the linear-time enumeration of superbubbles. Lemmas 3.1 and 3.2 were proved by Onodera et al. [OSS13] and Sung et al. [SSS+15], respectively.

**Lemma 3.1** ([OSS13])**.** *Any vertex can be the entrance (respectively exit) of at most one superbubble.*

Note that Lemma 3.1 does not exclude the possibility that a vertex is the entrance of a superbubble and the exit of another superbubble.

**Lemma 3.2** ([SSS+15])**.** *Let $G$ be a directed acyclic graph. We have the following two observations.*

*1) Suppose* $(\mathtt{p}, \mathtt{c})$ *is an edge in $G$, where* $\mathtt{p}$ *has one child and* $\mathtt{c}$ *has one parent, then* $\langle \mathtt{p}, \mathtt{c} \rangle$ *is a superbubble in $G$.*

*2) For any superbubble* $\langle s,t\rangle$ *in G, there must exist some parent* p *of* t *such that* p *has exactly one child* t.

**Remark 3.1.** *Consider a graph* $G = (\mathbb{V},\mathbb{E})$ *consisting of n vertices and* $n-1$ *edges such that* $\mathbb{V} = \{v_1, v_2, .., v_n\}$ *and* $\mathbb{E} = \{(v_{i-1}, v_i) \mid 1 < i \leq n\}$. *As a consequence of the first observation given in Lemma 3.2, G has the following superbubbles because each edge corresponds to a superbubble:* $\langle v_{i-1}, v_i\rangle \ \forall 1 < i \leq n$.

In this chapter, we start by showing another important property of superbubbles that is closely-related to Lemma 3.2.

**Lemma 3.3.** *For any superbubble* $\langle s,t\rangle$ *in a directed acyclic graph G, there must exist some child* c *of* s *such that* c *has exactly one parent* s.

**Proof.** Assume that all the children of s have more than one parent. Then, there must be some cycle or some child c which has a parent that does not belong to the superbubble $\langle s,t\rangle$. This is a contradiction. ■

### 3.2.1 Previously Best Algorithm

The algorithm by Sung et al. works in four steps [SSS+15] to identify all the superbubbles in a given directed graph $G$. These are as follows:

1. **Partition:** This step partitions the given graph into a set consisting of –

   a) subgraphs corresponding to each non-singleton strongly connected component (described in Subsection 2.4.3).

   b) a subgraph induced by the set of all the vertices involved in singleton strongly connected components.

   This step proceeds by finding all the strongly connected components of $G$. Then, for each non-singleton component (say $G_n$), two artificial vertices are added (if needed) – one acting as the source and the other as the sink for this component. Any outgoing edge from a vertex (say u) in $G_n$ to a vertex outside this component is replaced by an edge from u to the artificial sink. Similarly, any incoming edge from a vertex outside the component to a vertex u in this component is replaced by an edge from the artificial source to u. If there is no incoming (or outgoing) edge outside the component then there is no need to add the artificial source (or sink).

On the other hand, an artificial source is always added to the subgraph (say $G_s$) induced by singleton components. However, an artificial sink is added only if needed. Any incoming (or outgoing) edge from (or to) a vertex outside $G_s$ to (or from) a vertex, say u, is replaced by an edge from the artificial source (or u) to u (or sink), in the same fashion as done in a non-singleton strongly-connected component. Additionally, an edge is added from the artificial source to each of the original source vertices of $G_s$ (i.e. vertices with in-degree 0).

For the sake of clarity, in the subsequent steps and to avoid testing the existence of an artificial source, we introduce a minor modification with respect to the original algorithm in this step, which is to always include the artificial source. Thus in the case of a subgraph corresponding to a strongly connected component, if there is no incoming edge from a vertex outside the component then an edge from the artificial source to an arbitrary vertex (except the artificial sink) is added.

2. **Conversion to an acyclic subgraph:** This step creates a corresponding acyclic graph from a graph containing one strongly connected component such that both have the same superbubbles. This step will be executed for each subgraph corresponding to a non-singleton strongly connected component (say $G_n$). First, the recursive form of depth first search is run on $G_n$ starting with the artificial source to identify the back edges in the corresponding DFS-tree (as described in Subsection 2.4.1). Subsequently, this step transforms $G_n$ into an acyclic $G'_n$ as follows:

   Let s denote the artificial source and t denote the artificial sink if it exists in $G_n$. Each vertex u of $G_n$, except s and t (if it exists), is duplicated to create another copy, say u′ (the duplicate copy is being denoted using a "prime" added to the name of the vertex). All the vertices of $G_n$ as well as these newly created vertices (duplicates) constitute the vertex set of $G'_n$. The edge set of $G'_n$ comprises of edges added in accordance with the following rules:

   - Every edge of $G_n$ that involves s (i.e. (s,u)) is added without any change.
   - Every edge of $G_n$ that involves t leads to the addition of an edge between the duplicate copy of the vertex u (i.e. u′) and t. In other words, an edge (u,t) transforms into the edge (u′,t).
   - Every edge (u,v) in which u is not the artificial source and v is not the artificial sink leads to the addition of the following edges:

- If $(u, v)$ is not a back edge, the corresponding duplicate edges – $(u, v)$ and $(u', v')$ – are added.
- If $(u, v)$ is a back edge, an edge between the vertex $u$ and the duplicate of the vertex $v$ i.e. $v'$ is added. Thus, the cycle is broken since $G'_n$ has no edge $(u', v)$.

This step culminates with the addition of an artificial sink if it does not exist already, followed by the addition of edges from each vertex with no outgoing edge (out-degree 0) to this artificial sink.

3. **Identifying superbubbles in an acyclic subgraph:** Given an acyclic subgraph, this step consists of repeating the following test on each vertex $u$ in topological order (defined in Section 2.4.2), assuming that $u$ is the exit of some superbubble:

   Every vertex in the parent set of $u$ which has only $u$ as its child is 'merged' with $u$ till such a vertex exists and is not the last vertex in the *parent-list* (a list maintaining parents of $u$). At this point, if $p$ is the only vertex in the parent set of $u$ with only one child (which is $u$), then the vertices $p$ and $u$ form a superbubble. The superbubble $\langle p, u \rangle$ is reported in that case and the vertex $p$ is also merged with $u$. *Merging* some vertex $v$ with another vertex $u$, here, refers to merging the lists of the parents of the two vertices. Each vertex which exists in both the lists is added just once and its out-degree is adjusted by reducing it by one; vertex $v$ is removed from the parent list of $u$ and is deleted from the graph with the corresponding changes in the edges. Note that this step is dominated by the merging of parent-lists which can be done in $\mathcal{O}(m \log m)$ time for all the vertices if parent-lists are maintained as AVL trees [AL62] (see [SSS+15] for further details).

4. **Filtering:** In this step, valid superbubbles are extracted from the list of those reported by the previous step. Any superbubble with an artificial source as its entrance or an artificial sink as its exit is a 'spurious' superbubble. Similarly, a superbubble whose entrance is not from the original set of vertices (i.e. the entrance is some duplicate of a vertex created in Step 2) is a spurious superbubble. Therefore, we extract from the reported list of superbubbles only those superbubbles whose entrance is a vertex in the original graph, and test for its validity as follows:

- Superbubble $\langle u, v \rangle$ is valid if the vertex $u$ is an ancestor of the vertex $v$ in the DFS-tree of the subgraph and their duplicate vertices ($u'$ and $v'$) also form a superbubble.
- Superbubble $\langle u, v' \rangle$ is valid if the vertex $v$ (vertex whose duplicate is $v'$) is an ancestor of the vertex $u$ in the DFS-tree of the subgraph.

To summarise, this algorithm detects all the superbubbles in a given graph by first partitioning the graph using Step 1 and then for each subgraph executing Step 3 after turning it into an acyclic variant (if needed) using Step 2. Invalid and spurious superbubbles (generated due to the duplication step) are filtered out in the end.

## 3.3 Our Algorithm to find Superbubbles

As mentioned earlier, the bottleneck of the algorithm by Sung et al. is Step 3. All other steps can be executed in time that is linear in the size of the given graph. We propose an algorithm to improve this step – our main contribution is the proposed algorithm SUPERBUBBLE that reports *all* superbubbles in a directed acyclic graph $G = (\mathbb{V}, \mathbb{E})$ with exactly one source (vertex with in-degree 0) and exactly one sink (vertex with out-degree 0). For the sake of simplicity, for the rest of this chapter and in all the propositions, lemmas and theorems that follow, we use $G$ to denote a directed acyclic graph with exactly one source and exactly one sink, and we use $n$ and $m$ to denote the number of its vertices and edges respectively, that is, for $G = (V, E)$ we have $n = |V|$ and $m = |E|$.

### 3.3.1 An Overview

The algorithm SUPERBUBBLE starts by topologically ordering the vertices of graph $G$ and then identifying all the *candidates* of the possible entrances and exits of superbubbles according to Lemmas 3.2 and 3.3. The aim of this algorithm is accomplished with the help of the subroutine VALIDATESUPERBUBBLE, explained in the following section, which checks whether a given candidate $\langle s, t \rangle$ is a superbubble or not; if it is not, the algorithm returns an alternative entrance for a superbubble that ends at $t$, or $-1$ if it is clear that such an entrance does not exist.

### 3.3.2 Topological Ordering (ORD)

A *topological ordering* of $G$ (represented using an array ORD) maps each vertex to an integer between 1 and $n$, such that $\mathsf{ORD}[x] < \mathsf{ORD}[y]$ holds for all edges $(x, y) \in \mathbb{E}$. There exists a classical linear-time algorithm for computing the topological ordering of a directed acyclic graph as has been described in Subsection 2.4.2.

The subroutine TOPOLOGICALSORT, given below, is a simplified version that takes as input a single-source, single-sink directed acyclic graph, and produces a topological ordering of vertices. For the graph $G$ in Figure 3.1, TOPOLOGICALSORT produces the ordering given in Figure 3.2.

**Subroutine 3.1** TOPOLOGICALSORT : Computes the topological ordering (ORD) of the given $G$.

```
1: function TopologicalSort(G)                                   ▷
   Assumes that G is a directed acyclic graph with one source vertex, denoted source
2:     order ← n                                  ▷ a global variable
3:     for all (v ∈ 𝕍) do
4:         Visited[v] ← false
5:     end for
6:     RecursiveTopologicalSort(G, source)
7: end function

8: function RecursiveTopologicalSort(G, v)
9:     Visited[v]= true
10:    for all w ∈ 𝕍 adjacent to v do
11:        if Visited[w]= false then
12:            RecursiveTopologicalSort(G, w)
13:        end if
14:    end for
15:    ORD[v] ← order
16:    order ← order − 1
17: end function
```

Importantly, in this chapter we do not consider just any topological ordering of graph $G$ but only the one obtained by the subroutine TOPOLOGICALSORT. Note that this algorithm finds a directed spanning tree $\mathcal{T}$ of $G$ rooted at the source, which contains a path from the source to any vertex in $G$. The directed spanning tree $\mathcal{T}$ of $G$ obtained by the subroutine TOPOLOGICALSORT is presented by bold edges in

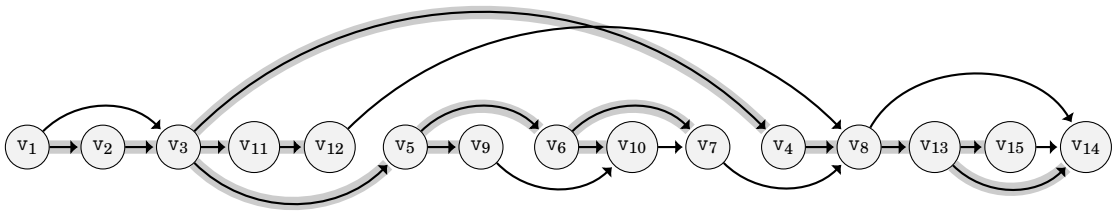


Figure 3.2: Vertices of Figure 3.1 in topological order, where $\mathsf{ORD}[v_1] = 1$, $\mathsf{ORD}[v_2] = 2$, $\mathsf{ORD}[v_3] = 3$, $\mathsf{ORD}[v_4] = 11$, $\mathsf{ORD}[v_5] = 6$, $\mathsf{ORD}[v_6] = 8$, $\mathsf{ORD}[v_7] = 10$, $\mathsf{ORD}[v_8] = 12$, $\mathsf{ORD}[v_9] = 7$, $\mathsf{ORD}[v_{10}] = 9$, $\mathsf{ORD}[v_{11}] = 4$, $\mathsf{ORD}[v_{12}] = 5$, $\mathsf{ORD}[v_{13}] = 13$, $\mathsf{ORD}[v_{14}] = 15$ and $\mathsf{ORD}[v_{15}] = 14$

Figure 3.2. It may be worth recalling that a directed rooted tree is also known as *arborescence*.

We next present a few important properties of the topological ordering obtained by the subroutine TOPOLOGICALSORT.

**Proposition 3.1.** *For any topological ordering ORD of vertices in graph G, if vertex* $u$ *is reachable from* $v$*, that is, if there is a path from* $v$ *to* $u$*, then* $\mathit{ORD}[v] < \mathit{ORD}[u]$.

**Proof.** If the path from $v$ to $u$ is of length 1, i.e., there is an edge $(v,u)$, then by the definition of topological ordering we have $\mathsf{ORD}[v] < \mathsf{ORD}[u]$. Otherwise, we denote the path from $v$ to $u$ of length $k$, $k > 1$, as $v, x_1, \ldots, x_{k-1}, u$. Then, by the definition of topological ordering we have $\mathsf{ORD}[v] < \mathsf{ORD}[x_1] < \cdots < \mathsf{ORD}[u]$. Transitively, we have $\mathsf{ORD}[v] < \mathsf{ORD}[u]$. ■

Note that $\mathsf{ORD}[v] < \mathsf{ORD}[u]$ does not imply that a path from $v$ to $u$ exists.

**Proposition 3.2.** *Let ORD be a topological ordering and* $\mathcal{T}$ *be a directed rooted spanning tree of graph G obtained by the subroutine* TOPOLOGICALSORT. *If there is a path in* $\mathcal{T}$ *from a vertex* $v$ *to a vertex* $u$*, then, for each vertex* $w$ *such that* $\mathit{ORD}[v] < \mathit{ORD}[w] < \mathit{ORD}[u]$*, there is a path from* $v$ *to* $w$.

**Proof.** Recall that $\mathcal{T}$ contains a path from the root to each vertex of the tree; this is also true for each subtree of $\mathcal{T}$. Furthermore, if there is a path from $v$ to $u$ in $\mathcal{T}$, then $u$ is contained in a subtree of $\mathcal{T}$ rooted at $v$, and each $w$ such that $\mathsf{ORD}[v] < \mathsf{ORD}[w] < \mathsf{ORD}[u]$ is also contained in the subtree rooted at $v$ (but not in the subtree rooted at $u$). Therefore, there is a path from $v$ to $w$, for each $w$ such that $\mathsf{ORD}[v] < \mathsf{ORD}[w] < \mathsf{ORD}[u]$. ■

We next show that in an ordering obtained by TOPOLOGICALSORT, a vertex has its topological ordering between the orderings of the entrance and the exit of a superbubble if and only if it belongs to the superbubble.

**Lemma 3.4.** *Let graph $G$ contain a superbubble $\langle s, t\rangle$. Then a topological ordering obtained by* TOPOLOGICALSORT *has the following properties.*

1. *For all* $x$ *such that* $x \in \mathbb{U}\backslash\{s,t\}$, $ORD[s] < ORD[x] < ORD[t]$.
2. *For all* $y$ *such that* $y \notin \mathbb{U}$, $ORD[y] < ORD[s]$ *or* $ORD[y] > ORD[t]$.

**Proof.** Recall that $\mathbb{U}$ is the set of vertices forming a superbubble (see Definition 3.1).

1. Since there is a path from the entrance $s$ of the superbubble to all $x \in \mathbb{U}\backslash\{s\}$, by Proposition 3.1 we have $ORD[s] < ORD[x]$ for all $x$ such that $x \in \mathbb{U}\backslash\{s\}$. Similarly, since there is a path from all $x \in \mathbb{U}\backslash\{t\}$ to the exit $t$ of the superbubble, by Proposition 3.1 we have $ORD[x] < ORD[t]$ for all $x$ such that $x \in \mathbb{U}\backslash\{t\}$. Therefore, for all $x$ such that $x \in \mathbb{U}\backslash\{s,t\}$, $ORD[s] < ORD[x] < ORD[t]$.

2. Suppose the contrary, that is, suppose that there exists some $y \notin \mathbb{U}$ such that $ORD[s] < ORD[y] < ORD[t]$. Since the superbubble $\langle s, t\rangle$ is itself a single-source, single-sink subgraph of $G$, any directed spanning tree of $G$ rooted at the source (i.e. vertex $s$), will contain a path from $s$ to $t$. Then, by Proposition 3.2 there also exists a path from $s$ to $y$ in $\mathscr{T}$ and thus also in $G$. However, by the definition of the superbubble, the only vertices reachable from $s$ without going through $t$ are the internal vertices of the superbubble — a contradiction. Therefore, for all $y$ such that $y \notin \mathbb{U}$, either $ORD[y] < ORD[s]$ or $ORD[y] > ORD[t]$.

■

### 3.3.3 Candidate List (*Candidates*)

The algorithm SUPERBUBBLE, after topologically ordering the vertices of graph $G$, checks each vertex in $\mathbb{V}$ in topological order to identify whether it is an exit or an entrance candidate (or both). According to Lemmas 3.2 and 3.3, a vertex $v$ is an exit candidate if it has at least one parent with exactly one child (out-degree 1) and an entrance candidate if it has at least one child with exactly one parent (in-degree 1). These identified entrance and exit candidates are stored in a doubly-linked list

| $j$ | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | $v_1$ | $v_2$ | $v_3$ | $v_{11}$ | $v_{12}$ | $v_5$ | $v_9$ | $v_6$ | $v_{10}$ | $v_7$ | $v_4$ | $v_8$ | $v_{13}$ | $v_{15}$ | $v_{14}$ |
| entrance | ✓ | | ✓ | ✓ | | ✓ | | | | | | ✓ | ✓ | | |
| exit | | | ✓ | | ✓ | | | | ✓ | ✓ | | ✓ | | | ✓ |

Figure 3.3: Candidate list for Figure 3.1. *Candidates* = [$v_1$(entrance), $v_3$(exit), $v_3$(entrance), $v_{11}$(entrance), $v_{12}$(exit), $v_5$(entrance), $v_{10}$(exit), $v_7$(exit), $v_8$(exit), $v_8$(entrance), $v_{13}$(entrance), $v_{14}$(exit)]. Note that both $v_3$ and $v_8$ appear twice in the list.

(represented as *Candidates*) ; specifically, an element of the list is a vertex along with a label specifying if it is an entrance or an exit candidate. Note that if a vertex $v$ is both an exit and an entrance candidate, then $v$ appears twice in the candidate list, first as an exit and then as an entrance (Figure 3.3). The elements of the candidate list are ordered according to ORD. There are at most $2n$ candidates, thus the cost of constructing a doubly-linked list of all the candidates is linear in $n$. In addition, each exit candidate in *Candidates* points to the nearest previous entrance candidate in the list. The candidate list of the graph in the running example has been shown in Figure 3.3.

### 3.3.4 Core of the Algorithm

Once the topological order ORD and the candidate list *Candidates* of graph $G$ have been computed, the algorithm SUPERBUBBLE (pseudo-code given as Algorithm 3.2) processes the candidates list in decreasing topological order (backwards). Let the list of candidates be $[v'_1, v'_2, \ldots, v'_\ell]$. The algorithm examines the candidates in decreasing order and does the following:

- If $v'_j$ is an entrance candidate, then delete $v'_j$ ;
- If $v'_j$ is an exit candidate, then the subroutine REPORTSUPERBUBBLE is called to find and report the superbubble ending at $v'_j$, that is, the superbubble $\langle v'_i, v'_j \rangle$, for some entrance candidate $v'_i$. REPORTSUPERBUBBLE also recursively finds and reports all the nested superbubbles between $v'_i$ and $v'_j$ with the help of recursive calls to itself.

For clarity of presentation, we next provide a list and a short description of the functions and arrays used by the algorithm SUPERBUBBLE and the subroutines

that it uses: REPORTSUPERBUBBLE and VALIDATESUPERBUBBLE. For the sake of simplicity, we use a vertex and its corresponding candidate (element in the candidate list) interchangeably. This does not add to the complexity of the algorithm as we can use an auxiliary array VerToCand, where VerToCand[$i$] stores a pointer to the element corresponding to the vertex $v_i$ in *Candidates* so as to provide a constant-time conversion from a vertex to the corresponding candidate.

1. ENTRANCE(v) takes as input a vertex v and outputs TRUE if v is an entrance candidate, that is, if it satisfies Lemma 3.3, and FALSE otherwise.

2. EXIT(v) takes as input a vertex v and outputs TRUE if v is an exit candidate, that is, if it satisfies Lemma 3.2, and FALSE otherwise.

3. INSERTENTRANCE(v) takes as input a vertex v, inserts it as a candidate at the end of *Candidates* and labels it as *entrance*.

4. INSERTEXIT(v) takes as input a vertex v, inserts it as a candidate at the end of *Candidates* and labels it as *exit*. In addition, it also stores a pointer for this exit candidate; the pointer points to the nearest entrance candidate appearing before this exit candidate in *Candidates*. Note the subtle consequence of the order of adding candidates – if v is also an entrance candidate, it is first added as an exit candidate and then as an entrance candidate. Therefore, the exit candidate corresponding to v will always point to some entrance candidate corresponding to a vertex other than v.

5. HEAD(*Candidates*) and TAIL(*Candidates*) return the first and the last element in *Candidates*, respectively.

6. DELETETAIL(*Candidates*) deletes the last element in *candidates*.

7. NEXT(v) returns the candidate following v in *candidates*.

8. PVSENTRANCECANDIDATE(v) takes as input a vertex v which is an exit candidate and returns the nearest entrance candidate appearing before this exit candidate in *Candidates*.

9. VERTEX($i$) returns the vertex that has the topological order $i$ i.e. outputs vertex v such that ORD[$i$] = v.

In addition to the above subroutines, the following arrays have been utilised explicitly.

1. The array ORD stores the topological order of the vertices.
2. The array PvsEntrance stores the previous entrance candidate s for each vertex v (v is not necessarily a candidate). Formally, PvsEntrance[v] = s, where s is an entrance candidate such that ORD[s] ≤ ORD[v] and there does not exist another entrance candidate $s'$ such that ORD[s] < ORD[$s'$] < ORD[v]. Note that in case v is an entrance candidate, PvsEntrance[v] = v. For instance, note that for the aforementioned example graph, PvsEntrance[$v_6$] = $v_5$ and PvsEntrance[$v_{13}$] = $v_{13}$.
3. The array AltEntrance is used to reduce the number of *entrance−exit* pairs that need to be considered as possible superbubbles. Array AltEntrance is further detailed in Subsection 3.4.1.

**Remark 3.2.** *It is also possible to design the algorithm for moving forward in the topological order instead of backwards.*

Note that the subroutine REPORTSUPERBUBBLE is called for each exit candidate in decreasing order either by the algorithm SUPERBUBBLE or through a recursive call to identify a nested superbubble. A call to REPORTSUPERBUBBLE(start, exit) checks the possible entrance candidates between start and exit, starting with the nearest previous entrance candidate (to exit). This task is accomplished with the help of the subroutine VALIDATESUPERBUBBLE, explained in the following section, which checks whether a given candidate superbubble ⟨s,t⟩ is a superbubble or not; if it is not, the algorithm returns either a "-1" which means that no superbubble ends at t, or an alternative entrance candidate for a superbubble that could end at t.

**Remark 3.3.** *We can avoid using the flag variable* found *by simply testing whether* s = valid. *This is because if an exit is tested with* start *itself, either* s *(valid superbubble) or -1 is returned by* VALIDATESUPERBUBBLE *(i.e. there can not be an alternative entrance with topological order less than that of the starting vertex* start*). However, use of the flag makes the correctness more explicit and clearer.*

**Algorithm 3.2** SUPERBUBBLE : Identifies superbubbles of the given directed acyclic graph $G$.

```
 1: function SUPERBUBBLE(G)
    ▷ Initialisation:
 2:     TOPOLOGICALSORT(G)
 3:     prevEnt ← null
 4:     for all v in topological order do
 5:         AltEntrance[v] ← null
 6:         if EXIT(v) then
 7:             INSERTEXIT(v)
 8:         end if
 9:         if ENTRANCE(v) then
10:             INSERTENTRANCE(v)
11:             prevEnt ← v
12:         end if
13:         PvsEntrance[v] ← prevEnt
14:     end for
    ▷ Main:
15:     while Candidates is not empty do
16:         if ENTRANCE(TAIL(Candidates)) then
17:             DELETETAIL(Candidates)
18:         else
19:             REPORTSUPERBUBBLE(HEAD(Candidates),TAIL(Candidates))
20:         end if
21:     end while
22: end function
```

For the graph $G$ in Figure 3.1, the algorithm SUPERBUBBLE makes exactly three calls to the subroutine REPORTSUPERBUBBLE:

1. REPORTSUPERBUBBLE($v_1, v_{14}$):
   First, it checks the exit candidate $v_{14}$ against the nearest previous entrance candidate, i.e. the vertex $v_{13}$. The call to VALIDATESUPERBUBBLE($v_{13}, v_{14}$) returns $v_8$ as an alternative entrance candidate. The new candidate is then checked and the superbubble $\langle v_8, v_{14}\rangle$ is reported.

2. REPORTSUPERBUBBLE($v_1, v_8$):
   First, it checks the exit candidate $v_8$ against the nearest previous entrance candidate, i.e. the vertex $v_5$. The call to VALIDATESUPERBUBBLE($v_5, v_8$) returns $v_3$ as an alternative entrance candidate. The new candidate is then

**Subroutine 3.3** REPORTSUPERBUBBLE : Reports the superbubble ending at `exit` (if any), including the nested ones.

```
1: function REPORTSUPERBUBBLE(start,exit)
2:     if start=null or exit=null or ORD[start] ≥ ORD[exit] then
3:         DELETETAIL(Candidates)
4:         return
5:     end if
6:     s ← PVSENTRANCECANDIDATE(exit)
7:     found = false
8:     while (ORD[s] ≥ ORD[start]) do
9:         valid ← VALIDATESUPERBUBBLE(s,exit)
10:        if valid = s then
11:            found = true
12:        end if
13:        if found or valid = AltEntrance[s] or valid = -1 then
14:            break
15:        end if
16:        AltEntrance[s] ← valid
17:        s ← valid
18:    end while
19:    DELETETAIL(Candidates)
20:    if found then
21:        REPORT(⟨s,exit⟩)
22:        while TAIL(Candidates) is not s do
23:            if EXIT(TAIL(Candidates)) then        ▷ Check for nested superbubbles
24:                REPORTSUPERBUBBLE(NEXT(s),TAIL(Candidates))
25:            else
26:                DELETETAIL(Candidates)
27:            end if
28:        end while
29:    end if
30:    return
31: end function
```

checked and the superbubble $\langle v_3, v_8 \rangle$ is reported. Additionally, two recursive calls are made:

a) REPORTSUPERBUBBLE($v_{11}, v_7$):
First, it validates $\langle v_5, v_7 \rangle$ and reports it. Then, it makes a recursive call – REPORTSUPERBUBBLE($v_{10}, v_{10}$) which terminates without reporting any superbubble.

b) REPORTSUPERBUBBLE($v_{11}, v_{12}$):
It validates$\langle v_{11}, v_{12}\rangle$ and reports it.

3. REPORTSUPERBUBBLE($v_1, v_3$):
It validates $\langle v_1, v_3\rangle$ and reports it.

## 3.4 Validating a Superbubble

In this section, we describe the subroutine VALIDATESUPERBUBBLE. The ability to validate a candidate superbubble depends on the following result related to the Range Minimum Query (mentioned in Subsection 2.3.3) problem.

In order to check whether a superbubble candidate $\langle s, t\rangle$ is a superbubble or not, we propose to utilise the range min/max query problem as follows:

- For a given graph $G = (\mathbb{V}, \mathbb{E})$ and for each vertex $v \in \mathbb{V}$ with topological order ORD[v], calculate the topological orders of the parent and the child of v that are topologically furthest from v.

$$\text{OutParent[ORD[v]]} = \min(\{\text{ORD[u]} \mid (u, v) \in \mathbb{E}\}),$$
$$\text{OutChild[ORD[v]]} = \max(\{\text{ORD[u]} \mid (v, u) \in \mathbb{E}\}).$$

- For an integer array A and indices $i$ and $j$ we define RANGEMIN(A, $i, j$) and RANGEMAX(A, $i, j$) to return the minimum and maximum values of A[$i..j$], respectively.

  Then, for a given superbubble candidate $\langle s, t\rangle$, where s and t are an entrance and an exit candidate respectively (satisfying Lemmas 3.1 and 3.2), if $\langle s, t\rangle$ is a superbubble then the following two conditions are valid:

$$\text{RANGEMIN(OutParent,ORD[s]+1,ORD[t])} = \text{ORD[s]},$$
$$\text{RANGEMAX(OutChild,ORD[s],ORD[t]-1)} = \text{ORD[t]}.$$

For example, Figure 3.4 represents both OutParent and OutChild arrays computed for the graph $G$ in Figure 3.1. Furthermore, a candidate $\langle v_5, v_8\rangle$ is not a superbubble as RANGEMIN(OutParent, ORD[$v_5$] + 1, ORD[$v_8$]) = 3 ≠ 6 = ORD[$v_5$].

It should be clear that after an $\mathcal{O}(n + m)$-time pre-processing, validating a superbubble requires $\mathcal{O}(1)$ time which is the cost for the range max/min query. The

| $j$ | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | $v_1$ | $v_2$ | $v_3$ | $v_{11}$ | $v_{12}$ | $v_5$ | $v_9$ | $v_6$ | $v_{10}$ | $v_7$ | $v_4$ | $v_8$ | $v_{13}$ | $v_{15}$ | $v_{14}$ |
| *OutParent*[$j$] | - | 1 | 1 | 3 | 4 | 3 | 6 | 6 | 7 | 8 | 3 | 5 | 12 | 13 | 12 |
| *OutChild*[$j$] | 3 | 3 | 11 | 5 | 12 | 8 | 9 | 10 | 10 | 12 | 12 | 15 | 15 | 15 | - |

Figure 3.4: OutParent and OutChild arrays for the graph in Figure 3.1.

subroutine VALIDATESUPERBUBBLE(`startVertex`, `endVertex`) is designed to return an appropriate entrance candidate for a superbubble ending at `endVertex` (if any), as shown in the pseudo-code Subroutine 3.4.

**Subroutine 3.4** VALIDATESUPERBUBBLE : Returns an appropriate entrance candidate for a superbubble ending at `endVertex` (if any).

```
1: function VALIDATESUPERBUBBLE(startVertex, endVertex)
2:     start ← ORD[startVertex]
3:     end ← ORD[endVertex]
4:     outChild ← RANGEMAX(OutChild, start, end − 1)
5:     outParent ← RANGEMAX(OutParent, start + 1, end)
6:     if outChild ≠ end then
7:         return -1
8:     end if
9:     if outParent = start then
10:        return startVertex
11:    else if ENTRANCE(VERTEX(outParent)) then
12:        return VERTEX(outParent)
13:    else
14:        return PvsEntrance[VERTEX(outParent)]
15:    end if
16: end function
```

An important observation is that a subsequent call to VALIDATESUPERBUBBLE, for a given entrance candidate, returns alternative entrance candidates in a strictly non-decreasing topological order as proved by Lemma 3.5.

**Lemma 3.5.** *Let* t *be the alternative entrance candidate returned by* VALIDATESUPERBUBBLE*(*s,e*). Then for any exit candidate* e′ *such that ORD*[s] < *ORD*[e′] < *ORD*[e]*, the order of the alternative entrance candidate* t′ *returned by* VALIDATESUPERBUBBLE*(*s,e′*) will be greater than or equal to the order of* t.

**Proof.** Recall that the alternative entrance t returned by the subroutine VALIDATESUPERBUBBLE(s,e) is either a vertex with topological order outParent, or the *previous entrance* of this vertex (given by PvsEntrance).

Since $\texttt{outParent} = \textsc{RangeMin}(\text{OutParent}, \text{ORD}[\texttt{s}]+1, \text{ORD}[\texttt{e}])$ and $\text{ORD}[\texttt{s}] < \text{ORD}[\texttt{e}'] < \text{ORD}[\texttt{e}]$, we have

$$\texttt{outParent}' = \textsc{RangeMin}(\text{OutParent}, \text{ORD}[\texttt{s}]+1, \text{ORD}[\texttt{e}'])$$

implying that $\texttt{outParent} \leq \texttt{outParent}'$. Therefore, $\text{ORD}[\texttt{t}] \leq \text{ORD}[\texttt{t}']$. ■

### 3.4.1 Validation and AltEntrance

In case the validation of the candidate pair $(\texttt{t}_0, \texttt{e})$ fails, VALIDATESUPERBUBBLE$(\texttt{t}_0, \texttt{e})$ returns either "-1" or an alternative candidate $\texttt{t}_1$ which might be an entrance of a superbubble ending at e. This alternative candidate $\texttt{t}_1$ is either a vertex $\texttt{u}_1$, if $\texttt{u}_1$ is an entrance candidate, or the previous entrance candidate of $\texttt{u}_1$ such that

$$\begin{aligned}\text{ORD}[\texttt{u}_1] &= \text{OutParent}[\text{ORD}[\texttt{v}_0]] \\ &= \textsc{RangeMin}(\text{OutParent}, \text{ORD}[\texttt{t}_0]+1, \text{ORD}[\texttt{e}]),\end{aligned}$$

where $\texttt{v}_0$ is some vertex between $\texttt{t}_0$ and e in the topological ordering.

Suppose $\texttt{t}_1$ is also not a valid entrance of the superbubble ending at e. Then, there must be a vertex $\texttt{v}_1$ such that $\text{ORD}[\texttt{t}_1] < \text{ORD}[\texttt{v}_1] < \text{ORD}[\texttt{t}_0]$, with some parent $\texttt{u}_2$ such that $\text{ORD}[\texttt{u}_2] = \text{OutParent}[\text{ORD}[\texttt{v}_1]]$. Then, the alternative entrance is some $\texttt{t}_2$, which is either a vertex $\texttt{u}_2$ or its previous entrance and thus $\text{ORD}[\texttt{t}_2] < \text{ORD}[\texttt{t}_1]$. A series of such failed validations produces a sequence $\texttt{t}_1, \texttt{t}_2, \ldots$ of failed alternative entrance candidates.

A notable observation here is that any entrance $\texttt{t}_i$, for $i \geq 1$, from such a sequence is an invalid entrance not only for the superbubble ending at e but also for all those ending at any other exit vertex $e'$ such that $\text{ORD}[\texttt{t}_{i-1}] < \text{ORD}[\texttt{e}'] < \text{ORD}[\texttt{e}]$ and $\texttt{t}_i = \text{VALIDATESUPERBUBBLE}(\texttt{t}_{i-1}, \texttt{e}')$. This is the case because the vertex $\texttt{v}_i$, which causes the alternative entrance $\texttt{t}_i$ to fail, is such that $\text{ORD}[\texttt{t}_i] < \text{ORD}[\texttt{v}_i] < \text{ORD}[\texttt{t}_{i-1}]$ for $i \geq 1$. In other words, if $\texttt{t}_i$ failed due to some $\texttt{v}_i$ when tested with e then $\texttt{v}_i$ will also be the reason for failure whenever $\texttt{t}_i$ is tested with any exit candidate between $\texttt{t}_{i-1}$ and e.

This is where the array AltEntrance plays an important role: using AltEntrance to store AltEntrance[$t_{i-1}$] = $t_i$ for $i \geq 1$ enables us to skip this sequence at a later stage if $t_i$ is returned by the subroutine VALIDATESUPERBUBBLE($t_{i-1}$, e′).

## 3.5 Analysis of the Algorithm

In this section, we analyse the correctness, the running time, and the space requirement of the proposed algorithm SUPERBUBBLE.

### 3.5.1 Correctness and Time Complexity

For simplicity, in the following lemma we will slightly abuse the terminology and refer to ⟨s, t⟩ as a *superbubble* if it satisfies the first three conditions given in Definition 3.1, and as a *minimal superbubble* if it also satisfies the last condition in the same definition.

**Lemma 3.6.** *For a given exit candidate* e*, let* s *be the entrance candidate such that superbubble* ⟨s, e⟩ *is reported by the subroutine* VALIDATESUPERBUBBLE*(*s,e*). Then* ⟨s, e⟩ *is a minimal superbubble.*

**Proof.** By contradiction, let e′ be an exit candidate such that ⟨s, e′⟩ is also a superbubble and ORD[s] < ORD[e′] < ORD[e]. Then, either ORD[e] = ORD[e′] + 1 or there is at least one vertex v such that ORD[e′] < ORD[v] < ORD[e].

In the first case, ORD[e] = ORD[e′] + 1 implies that e is the only child of e′ and e′ is the only parent of e, which, by Lemma 3.2 makes ⟨e′, e⟩ a superbubble.

In the second case also, where there is at least one vertex v such that ORD[e′] < ORD[v] < ORD[e], we argue that ⟨e′, e⟩ must be a superbubble. Indeed, ⟨e′, e⟩ satisfies the following conditions:

1. **Reachability:** Since ⟨s, e⟩ is a superbubble, e is reachable from s. If ⟨s, e′⟩ is also assumed to be a superbubble, any path from s to e must go through e′, therefore e is reachable from e′.

2. **Matching:** The only vertices reachable from e′ without going through e are those whose topological order is between ORD[e′] and ORD[e]. Indeed, since ⟨s, e⟩ and ⟨s, e′⟩ are superbubbles, all these vertices are reachable from s through e′, and no vertices with topological orders greater than ORD[e] are

reachable from $e'$ without going through $e$. Similarly, there are no edges between vertices with topological orders less than $\mathsf{ORD}[e']$ and those with topological orders between $\mathsf{ORD}[e']$ and $\mathsf{ORD}[e]$. Therefore, the only vertices from which $e$ is reachable without going through $e'$ are those whose topological orders are between $\mathsf{ORD}[e']$ and $\mathsf{ORD}[e]$.

3. **Acyclicity:** Since $G$ is acyclic and $\langle e', e\rangle$ is its subgraph, it is also acyclic.

In both the cases, due to the fact that for each exit candidate the entrance candidates are checked in reverse topological order, VALIDATESUPERBUBBLE would have been called on $\langle e', e\rangle$ first, and would have reported $\langle e', e\rangle$ instead of $\langle s, e\rangle$. Therefore, $\langle s, e\rangle$ is a minimal superbubble. ■

**Lemma 3.7.** *For the given entrance and exit candidates* $s$ *and* $t$*, respectively, the subroutine* VALIDATESUPERBUBBLE *reports* $\langle s, t\rangle$ *if and only if* $\langle s, t\rangle$ *is a superbubble.*

**Proof.** We prove the lemma by showing that if $\langle s, t\rangle$ is a superbubble then the subroutine VALIDATESUPERBUBBLE reports it, and if VALIDATESUPERBUBBLE reports $\langle s, t\rangle$ then $\langle s, t\rangle$ is a superbubble.

1. We start by showing that if $\langle s, t\rangle$ is a superbubble then it is reported by the subroutine VALIDATESUPERBUBBLE. Indeed, by Lemma 3.4, all the vertices with topological orderings between $s$ and $t$ belong to the superbubble $\langle s, t\rangle$. Therefore, the minimum $\mathsf{OutParent}$ is $s$ and the maximum $\mathsf{OutChild}$ is $t$ and thus VALIDATESUPERBUBBLE reports $\langle s, t\rangle$.

2. We next show that if the subroutine VALIDATESUPERBUBBLE reports $\langle s, t\rangle$ then $\langle s, t\rangle$ is a superbubble. Let $\mathtt{start}$ and $\mathtt{end}$ be two integers, such that $\mathsf{ORD}[s] = \mathtt{start}$ and $\mathsf{ORD}[t] = \mathtt{end}$. The graph $G$, as defined, has a single source $r$ and a single sink $r'$; this implies that any vertex $v \in \mathbb{V}$ is reachable from the source $r$ and, at the same time, can reach the sink $r'$. This is also true for $s$, $t$, and for any vertex $v$ such that $\mathsf{ORD}[s] < \mathsf{ORD}[v] < \mathsf{ORD}[t]$.

   First, we show that $t$ is **reachable** from $s$. Recall that $t$ is an exit candidate, so it has a parent $p$ with out-degree 1. Assume that $t$ is not reachable from $s$. Then, there must be a path from $r \rightsquigarrow t$ which does not involve $s$. This implies that either $\mathsf{OutParent}[\mathtt{end}] < \mathtt{start}$, or there exists a vertex $v$ such

that start < ORD[v] < end, OutParent[v] < start and there exists a path $r \rightsquigarrow v \rightsquigarrow t$, which is a contradiction.

Similarly, we can show that every vertex v such that start < ORD[v] < end satisfies the **matching** criterion of the superbubble.

The **acyclicity** criterion is guaranteed by the acyclicity of $G$ and the **minimality** is satisfied by the design of subroutine REPORTSUPERBUBBLE which assigns each exit of a superbubble to the nearest entrance, and by Lemma 3.6.

■

**Lemma 3.8.** *For a given exit candidate* e*, let* t *be the alternative entrance candidate returned by the subroutine* VALIDATESUPERBUBBLE*(*s,e*). Then any entrance candidate between* t *and* e *cannot be a valid entrance for the superbubble ending at* e.

**Proof.** By contradiction, assume that $s'$ is an entrance candidate between t and e such that $\langle s', e \rangle$ is a superbubble. If $s'$ had been between s and e, it would have already been reported, as SUPERBUBBLE checks entrance candidates in reverse topological order starting from e. Therefore, $s'$ is between t and s, such that ORD[t] < ORD[$s'$] < ORD[s] < ORD[e].

Let outParent = RANGEMIN(OutParent, ORD[s] + 1, ORD[e]). Then, the vertex at outParent is between t and $s'$, otherwise VALIDATESUPERBUBBLE(s,e) would have returned $s'$ (instead of t). Therefore, ORD[t] ≤ outParent < ORD[$s'$].

Let outParent$'$ = RANGEMIN(OutParent, ORD[$s'$]+1, ORD[e]). Then outParent$'$ ≤ outParent. This implies that outParent$'$ ≤ outParent < ORD[$s'$]. However, for $\langle s', e \rangle$ to be a valid superbubble, outParent$'$ should have been equal to ORD[$s'$]. Hence, the assumption is wrong and thus it is proved that there cannot be an entrance candidate between t and e, which is a valid entrance for the superbubble ending at e. ■

**Lemma 3.9.** *For the given entrance and exit candidates* s *and* $e_1$*, respectively, let* AltEntrance[s] *be set to* $t_1$ *which later gets reset to* $t_2$ *(such that* $t_2 \neq t_1$*) while considering* s *with another exit candidate* $e_2$*. Then, no exit candidate between* s *and* $e_2$ *can reset* AltEntrance[s] *to* $t_1$ *again.*

**Proof.** Let $e_3$ be an exit candidate between s and $e_2$ such that the call to the subroutine VALIDATESUPERBUBBLE(s, $e_3$) returns $t_3$. Then, by Lemma 3.5, ORD[$t_1$] ≤

$\mathsf{ORD}[\mathtt{t}_2] \leq \mathsf{ORD}[\mathtt{t}_3]$. Since $\mathtt{t}_1 \neq \mathtt{t}_2$, we have $\mathsf{ORD}[\mathtt{t}_1] < \mathsf{ORD}[\mathtt{t}_2] \leq \mathsf{ORD}[\mathtt{t}_3]$. Therefore, $\mathsf{ORD}[\mathtt{t}_1] < \mathsf{ORD}[\mathtt{t}_3]$ and $\mathsf{AltEntrance}[\mathtt{s}]$ cannot be reset to the same value $\mathtt{t}_1$ again. ■

**Theorem 3.1.** *The algorithm* SUPERBUBBLE *reports all superbubbles, and only superbubbles, in graph G in decreasing topological order of their exit vertices in* $\mathcal{O}(n+m)$ *time.*

**Proof.** Consider an execution of SUPERBUBBLE. Let $[\langle \mathtt{s}_1, \mathtt{t}_1 \rangle, \cdots, \langle \mathtt{s}_k, \mathtt{t}_k \rangle]$ be the list of successive superbubbles reported just after the execution of Line 21 of the subroutine REPORTSUPERBUBBLE, where $\mathsf{ORD}[\mathtt{t}_1] > \mathsf{ORD}[\mathtt{t}_2] > \cdots > \mathsf{ORD}[\mathtt{t}_k]$.

1. First, we show that each $\langle \mathtt{s}_i, \mathtt{t}_i \rangle$ reported by the algorithm in Line 21 is a superbubble. This follows from Lemma 3.7.

2. Second, no superbubble is missed out by the algorithm as proved by the following arguments. The subroutine REPORTSUPERBUBBLE is called for each exit candidate in decreasing order. The entrance candidate for the superbubble (if any) ending at `exit` will only be between `start` and `exit`, where `start` is either the head of the the candidates list (when subroutine REPORTSUPERBUBBLE is called from the algorithm SUPERBUBBLE) or the next candidate of the entrance of an outer superbubble (when called through a recursive call to identify a nested superbubble). A call to the subroutine REPORTSUPERBUBBLE(`start`, `exit`) checks the possible entrance candidates between `start` and `exit`, starting with the nearest previous entrance candidate (to `exit`). The subroutine VALIDATESUPERBUBBLE either successfully validates an entrance candidate, or returns a “-1", or returns an alternative entrance candidate. From Lemma 3.8, there cannot be any valid entrance between this alternative entrance and `exit`. If this alternative entrance starts a sequence of entrances already checked for some exit candidate previously (as depicted by the array AltEntrance), then all entrances of that sequence will be skipped, otherwise this alternative entrance will be tested. However, as mentioned in Subsection 3.4.1, none of the entrance candidates in the skipped sequence can be valid. Therefore, for each exit candidate, every potential entrance candidate is checked for validity, and those which are not considered are not valid.

3. Third, the running time of SUPERBUBBLE is $\mathcal{O}(n+m)$. Indeed, the running time of the TOPOLOGICALSORT and computing the candidates list is $\mathcal{O}(n+m)$. Furthermore, all list operations cost constant time each, and sum up to a linear cost of $\mathcal{O}(n)$, as there are at most $2n$ candidates in the list. Finally, each call to VALIDATESUPERBUBBLE costs $\mathcal{O}(1)$. The total number of times VALIDATESUPERBUBBLE is called is $\mathcal{O}(n+m)$. This is because the subroutine VALIDATESUPERBUBBLE is called once for each exit candidate from REPORTSUPERBUBBLE, and the total number of such calls is bounded by $\mathcal{O}(n)$. Additionally, it is called every time a new *alternative entrance* sequence is generated by the subroutine VALIDATESUPERBUBBLE. It follows from Lemma 3.9 that once an AltEntrance sequence is reset, it cannot be generated again by subsequent calls to the subroutine VALIDATESUPERBUBBLE. This resetting of AltEntrance for each entrance candidate (Line 16) thus enables avoiding repeated checks of the same sequences of entrance candidates. Resetting is done every time an edge is considered for the first time between a vertex (in between an entrance candidate `startVertex` and an exit candidate `endVertex`) and its topologically furthest parent (whose order is less than that of `startVertex`). Thus, the total number of times AltEntrance will be reset (for all the entrance candidates) is bounded by $\mathcal{O}(m)$.

   Therefore, the total running time for reporting all superbubbles in the graph $G$ is $\mathcal{O}(n+m)$.

■

### 3.5.2 Space Complexity

It is trivial to see that the overall space consumed by the algorithm is linear with respect to the size of the graph. Moreover, the graph in not needed in memory after the initialisation stage. In fact, the working memory requirement of the algorithm is $\mathcal{O}(n)$ – the size of every auxiliary array and data structure used is $n$ and the size of the candidate list is $\leq 2n$.

## 3.6 Impact

The theoretical impact of the proposed algorithm is the improvement in the bottleneck stage of the pipeline for reporting the superbubbles in a general directed graph,

making the overall algorithm an optimal one. From a practical view-point, we noted that no implementation of the previous state-of-the-art algorithm was available and consequently we implemented the whole pipeline as a software tool (using the proposed algorithm for Stage 3 and algorithm by Sung et al. for the stages 1, 2, and 4).

The software tool was picked up by the scientific community working on genome assembly as soon as it was made available and is currently being used in the *vg (variation graph) tool-kit* developed by *Richard Durbin's lab at the Wellcome Trust Sanger Institute*. Furthermore, the proposed algorithm spurred-on generalisations of superbubbles – termed as ***Snarls*** and ***Ultrabubbles*** – for bidirected and biedged graphs [PNGH17].

CHAPTER 4

# ELASTIC-DEGENERATE STRINGS

Motivated by applications like intra-species genomic variation studies, in this chapter, we extend the notion of gapped strings to *elastic-degenerate strings*. An elastic-degenerate string can be seen as an ordered collection of solid (standard) strings interleaved by *elastic-degenerate symbols*; each such symbol corresponds to a set of two or more variable-length solid strings. In this chapter, we present an algorithm for solving the pattern matching problem with a solid pattern and an elastic-degenerate text running in $\mathcal{O}(N + \alpha\gamma mn)$ time, where $m$ is the length of the pattern, $n$ and $N$ are the length and total size of the elastic-degenerate text, respectively, $\alpha$ and $\gamma$ are parameters, respectively representing the maximum number of strings in any elastic-degenerate symbol of the text and the maximum number of elastic-degenerate symbols spanned by any occurrence of the pattern in the text. The space used by the algorithm is linear in the size of the input for a constant number of elastic-degenerate symbols in the text.

The chapter is organised as follows: we begin by giving the background of the problem and discussing related work in the literature in Section 4.1. In Section 4.2, we introduce the basic definitions and formalise the notions of elastic-degeneracy that will be used throughout. We delineate the algorithm in Section 4.3 and present its analysis in Section 4.4. The experimental results are described in Section 4.5. Finally, the chapter concludes with Section 4.6 wherein we mention the impact this proposed model has created.

## 4.1 Background

In many applications like molecular biology (where sequences are considered as strings over a fixed size alphabet $\Sigma$), if the specific nature of data is to be accommodated, we are required to allow some positions in the sequence to contain, instead of a single letter from $\Sigma$, a subset of $\Sigma$. Such *degenerate (indeterminate)* symbols can be interpreted to mean that the exact letter at the given position is not known, but is suspected to be one of the specified letters. For example, the string $[\begin{smallmatrix}\mathtt{a}\\\mathtt{c}\end{smallmatrix}]\mathtt{ac}[\begin{smallmatrix}\mathtt{b}\\\mathtt{c}\end{smallmatrix}]\mathtt{a}[\begin{smallmatrix}\mathtt{b}\\\mathtt{a}\\\mathtt{c}\end{smallmatrix}]$ is a degenerate string of length 6 over $\Sigma = \{\mathtt{a},\mathtt{b},\mathtt{c}\}$; the positions 1, 4, and 6 are *non-solid* positions because these can be occupied by any of the symbols from the specified set e.g. either of the symbols – a or c – may occur at the first position.

In biological sequences, a position in one string may match with various symbols in other strings. Pattern matching in degenerate strings is particularly relevant in the context of coding biological sequences. Due to the degeneracy of the genetic code, two dissimilar DNA sequences can be translated into identical protein sequences. Without taking this degeneracy into account, many associations between biological entities can be overlooked. For example, the following six DNA codons are all translated into the amino acid *Leucine*: TTA, TTG, CTT, CTC, CTA and CTG. This example highlights the significance of solving problems relating to degeneracy in strings. Please refer to Subsection 2.5.1 for a general introduction to the concepts of translation, codons, genetic code etc.

A more restrictive variant of degenerate strings – which allows at a given position a subset consisting of either a single letter or all the letters of $\Sigma$ – was proposed by Fischer and Paterson in their seminal work [FP74]. For example, $\mathtt{ab}\diamond\mathtt{ac}$ is an instance of a string of length 5 where the third position carries a *hole* or *don't care* or *wild card* symbol (usually represented by $\diamond$ or $*$) which can match any letter from the alphabet. This restrictive model has been called "partial words" or "strings with wild cards/holes/don't cares" in recent years. It has been considered for various classical problems, other than the pattern matching problem, that involve structured regularities in strings like covers, periods etc.; [BS12] presents a comprehensive survey on partial words. However, this model is not as expressive as degenerate strings when it comes to capturing the uncertainty in biological sequences.

The pattern matching problem in degenerate strings was first proposed by Abrahamson as "generalised string matching" in 1987 [Abr87] along with an algorithm which, however, was not efficient enough to be used in practice. Subse-

quently, more practically-efficient algorithms were proposed in 1992 using the bit-mapping approach (the so-called "Shift-Or" technique) [BYG92, WM92]). The bit-mapping approach reduces a problem to bit operations (Shift, AND, OR etc.) over bit-vectors and exploits the parallelism of those operations over bits in a computer word. Some of the more efficient and practical algorithms for pattern matching on degenerate strings developed over the last decade can be found in [HSW08, IMR08, SW09, CIK$^+$16b]. These are based on disparate techniques like the Sunday variant [Sun90] of Boyer–Moore pattern-matching algorithm [BM77], Fast Fourier Technique [FP74], Landau–Vishkin's algorithm for approximate matches [LV89]. Moreover, numerous studies comprising of algorithmic and combinatorial perspectives for solving a range of problems involving degenerate strings have enriched the literature since then; see the recent woks presented in [CIK$^+$17, BSBDW17] and references therein for regularity related problems on degenerate strings. Furthermore, another variant of degenerate strings – weighted strings – which additionally associate a probability of occurrence (weight) to each letter in some non-solid position, have given rise to another line of research in the context of degeneracy. For example, [BP18, KPR16, BKPR16] present the recent algorithmic advances in the problems related to pattern matching, structured regularities, indexing etc. in the weighted strings' setting.

Moving on to another such representation for characterising uncertainty in sequential data (strings), we have a *gapped string* (or *compound pattern* or *composite pattern*). As mentioned in Chapter 1, a gapped string is an ordered collection of standard strings (*seeds*) separated by variable-length *gaps* defined by an ordered collection of intervals [CS04]. Following the representation used in [RIL$^+$06], the string $X = \mathtt{ab}*^{2,4}\mathtt{aab}$ is a gapped string with two seeds interspersed by one gap of size in the range 2 to 4; the gap represents any string of length between the specified range. Here, for $\Sigma = \{\mathtt{a},\mathtt{b}\}$, any string of length 2, 3, or 4 will match the gap; each string corresponding to the gap should be preceded by the string $\mathtt{ab}$ and followed by the string $\mathtt{aab}$ to match the gapped string $X$. A gapped string corresponds to the notion of a "structured motif" used in molecular biology. A single motif is simply a conserved DNA (or RNA) sequence; a structured motif consists of two or more single motifs separated by possibly variable length "spacers" (gaps). Extracting and identifying specified structured motifs in DNA sequences is of particular interest because they model the functional combinations of transcription factor binding sites (TFBS) for co-regulated genes [EP02, CFOS04]. Transcription and TFBS have been

described in Subsection 2.5.1.

The problem of pattern matching and discovery in the context of gapped strings and its variants has been studied extensively using combinatorial approaches. In [Pis14], Pissis presented a high performance computing tool for structured motif extraction from the large datasets along with a review of other algorithms/tools in detail; a survey of the algorithms for pattern matching with gaps has been provided in [WQX14]. Recently, Alatabbi et al. presented a fast and simple algorithm in [AAH$^+$15] which is based on another approach suggested independently in [MPVZ05] and [RIL$^+$06] wherein the presented algorithm progresses in two phases – finding the occurrences of seeds followed by attempting to stitch them together considering the gap constraints. The weaknesses of some of the other notable approaches – using regular expression or bit parallelism – have been argued in [BLGVW10].

**Our Contribution**. Here we propose a model to capture the macro-level uncertainty in sequential data— *elastic-degenerate strings*— a *hybrid* of gapped strings and degenerate strings. An elastic-degenerate string can be visualised as an ordered collection of $k > 1$ strings interleaved by $k-1$ elastic-degenerate symbols. For instance, $\texttt{aab}\begin{bmatrix}\texttt{bb}\\ \texttt{aab}\end{bmatrix}\texttt{cab}\begin{bmatrix}\texttt{abcab}\\ \texttt{cba}\\ \texttt{aca}\end{bmatrix}\texttt{bac}$ is an example of an elastic-degenerate string over $\Sigma = \{\texttt{a},\texttt{b},\texttt{c}\}$.

This generalisation of the concept of *degeneracy* is motivated by several data mining problems [LBKP14] which can be reduced to the core task of discovering occurrences of one or more patterns in a text that can best be described as an ordered collection of strings interleaved by sets of variable-length strings. In the specific case of genomics, a representation that encodes a set of related genomes with variations in the reference genome itself (called the Population Reference Genome in [MdOEMI16]), can be seen as an elastic-degenerate string.

Summing up, a gapped string, which specifies the constraint on only the length of the gap between two consecutive seeds, differs from an elastic-degenerate string because only the latter precisely defines the possible strings (of varying lengths) that can exist between those seeds. However, this precise identification of 'allowed' strings in a gap makes the pattern matching problem, in the context of elastic-degenerate strings, algorithmically more challenging.

## 4.2 Preliminaries

In this section, we give the terminology to build the concept of elastic-degeneracy by presenting the following definitions and examples. For the purpose of clearly distinguishing the text or the pattern string and, in turn, to maintain consistency throughout the chapter, we will use capital letters to denote strings in this chapter.

**Definition 4.1** (Seed: $S$)**.** A *seed* $S$ is a (possibly empty) string over $\Sigma$.

**Definition 4.2** (Elastic-Degenerate Symbol: $\xi$)**.** An *elastic-degenerate symbol* $\xi$, over a given alphabet $\Sigma$, is a set of two or more distinct strings over $\Sigma$ (i.e. $\xi \subseteq \Sigma^*$ and $|\xi| > 1$). An elastic-degenerate symbol $\xi$ is denoted by $\begin{bmatrix} E_1 \\ E_2 \\ \vdots \\ E_{|\xi|} \end{bmatrix}$, where each $E_i,\ 1 \leq i \leq |\xi|$, is a solid string.

**Definition 4.3** (Elastic-Degenerate String: $\hat{X}$)**.** An *elastic-degenerate string* $\hat{X}$, over a given alphabet $\Sigma$, is a sequence $S_1\xi_1S_2\xi_2S_3\ldots S_{k-1}\xi_{k-1}S_k$, where $S_i,\ 1 \leq i \leq k$, is a seed and $\xi_i,\ 1 \leq i \leq k-1$ is an elastic-degenerate symbol.

An elastic-degenerate string $\hat{X}$ can be visualised as follows:

$$\hat{X} = S_1 \begin{bmatrix} E_{1,1} \\ E_{1,2} \\ \vdots \\ E_{1,|\xi_1|} \end{bmatrix} S_2 \begin{bmatrix} E_{2,1} \\ E_{2,2} \\ \vdots \\ E_{2,|\xi_2|} \end{bmatrix} S_3 \ldots S_{k-1} \begin{bmatrix} E_{k-1,1} \\ E_{k-1,2} \\ \vdots \\ E_{k-1,|\xi_{k-1}|} \end{bmatrix} S_k.$$

**Example 4.1.** $\hat{X} = \texttt{abbc} \begin{bmatrix} \texttt{ab} \\ \texttt{cab} \\ \texttt{acca} \end{bmatrix} \texttt{cca} \begin{bmatrix} \texttt{aabcab} \\ \texttt{cba} \end{bmatrix} \texttt{bb}$ *is an elastic-degenerate string, where we have the following:*

- *Three seeds:* $S_1 =$ abbc*,* $S_2 =$ cca*, and* $S_3 =$ bb.
- *Two elastic-degenerate symbols:*
  $\xi_1 = \begin{bmatrix} \texttt{ab} \\ \texttt{cab} \\ \texttt{acca} \end{bmatrix}$ *and* $\xi_2 = \begin{bmatrix} \texttt{aabcab} \\ \texttt{cba} \end{bmatrix}$.
- *For* $\xi_1$*:* $E_{1,1} =$ ab*,* $E_{1,2} =$ cab*,* $E_{1,3} =$ acca.
- *For* $\xi_2$*:* $E_{2,1} =$ aabcab*,* $E_{2,2} =$ cba.

Observe the use of $\hat{X}$ to distinguish an elastic-degenerate string from a solid string $X$ or a degenerate string $\tilde{X}$. Further, we will be using $\xi$ to denote an elastic-degenerate symbol and $E_{i,j}$ to denote a string from the set representing elastic-degenerate symbol $\xi_i$ in a string. In the following, we define three characteristics of a given elastic-degenerate string $\hat{X}$ with $k$ seeds.

**Definition 4.4** (Total Size: $\|\hat{X}\|$)**.** The *total size* of $\hat{X}$, denoted by $\|\hat{X}\|$, is defined as the sum of the total length of its seeds and the total length of all the strings in each of its elastic-degenerate symbols:

$$\|\hat{X}\| = \sum_{i=1}^{k} |S_i| + \sum_{i=1}^{k-1} \sum_{j=1}^{|\xi_i|} |E_{i,j}|$$

**Definition 4.5** (Length: $|\hat{X}|$)**.** The *length* of $\hat{X}$, denoted by $|\hat{X}|$, is defined as the sum of the total length of its seeds and the total number of its elastic-degenerate symbols:

$$|\hat{X}| = \sum_{i=1}^{k} |S_i| + k - 1$$

Informally, the total number of positions in $\hat{X}$ is its length considering an elastic-degenerate symbol to occupy only one position. Intuitively, a position belonging to some seed will be called a *solid position* and that of an elastic-degenerate symbol will be called an *elastic-degenerate position*. In the running example, the total length of the seeds is 9, hence, $\|\hat{X}\| = 9 + (2 + 3 + 4) + (6 + 3) = 27$, while $|\hat{X}| = 9 + 2 = 11$. The first a occurs at (solid) position 1, followed by b at (solid) position 2 and so on. $\xi_1$and $\xi_2$ are at (elastic-degenerate) positions 5 and 9, respectively and the last b is at (solid) position 11. As in case of a solid string, a **factor** of $\hat{X}$ (represented as $\hat{X}[i..j],\ 1 \leq i \leq j \leq n$) is the sequence $\hat{X}[i]\hat{X}[i+1]\hat{X}[i+2]..\hat{X}[j]$ (i.e. the contiguous *chunk* from the position $i$ to the position $j$).

**Definition 4.6** (Possibility-Set: $\Re$)**.** The Possibility-set $\Re$ for the elastic-degenerate string

$$\hat{X} = S_1\xi_1S_2\xi_2S_3\ldots S_{k-1}\xi_{k-1}S_k$$

is defined as follows:

$$\Re = \{S_1E_{1,r_1}S_2E_{2,r_2}\ldots E_{k-1,r_{k-1}}S_k\} \ \ \forall r_i, 1 \leq i \leq k-1 \text{ such that } 1 \leq r_i \leq |\xi_i|.$$

Informally, the *possibility-set* $\Re$ of $\hat{X}$ is the set of all possible solid strings obtained from $\hat{X}$. A solid string can be obtained by replacing each of the elastic-degenerate symbols with one of its constituent strings. In the running example, $\Re$ =

{abbcabccaaabcabbb, abbcabccacbabb, abbccabccaaabcabbb, abbcabccacbabb, abbcaccaccaaabcabbb, abbcaccaccacbabb}. Note that constituent strings replacing the elastic-degenerate symbols have been underlined for clarity.

We are now in a position to define *matching* and *occurrence* in the context of elastic-degenerate strings.

**Definition 4.7** (Matching)**.** An elastic-degenerate string $\hat{X}$ with $k$ seeds and a solid string $Y$ are said to match, denoted by $\hat{X} \simeq Y$, if, and only if, there exists a solid string $S = S_1 E_{1,r_1} S_2 E_{2,r_2} \ldots E_{k-1,r_{k-1}} S_k$, $1 \leq r_i \leq |\xi_i|$, obtained from $\hat{X}$ (i.e. $S \in \Re$ of $\hat{X}$), such that $S = UYV$, where $U, V \in \Sigma^*$, satisfying:

$$\begin{cases} U = \varepsilon,\ V = \varepsilon & \text{if } S_1 \neq \varepsilon,\ S_k \neq \varepsilon \\ E_{1,r_1} \neq \varepsilon,\ V = \varepsilon,\ U \text{ is either empty or a proper prefix of } E_{1,r_1} & \text{if } S_1 = \varepsilon,\ S_k \neq \varepsilon \\ E_{k-1,r_{k-1}} \neq \varepsilon,\ U = \varepsilon,\ V \text{ is either empty or a proper suffix of } E_{k-1,r_{k-1}} & \text{if } S_1 \neq \varepsilon,\ S_k = \varepsilon \\ E_{1,r_1} \neq \varepsilon,\ U \text{ is either empty or a proper prefix of } E_{1,r_1}, \\ \quad E_{k-1,r_{k-1}} \neq \varepsilon,\ V \text{ is either empty or a proper suffix of } E_{k-1,r_{k-1}} & \text{if } S_1 = \varepsilon,\ S_k = \varepsilon. \end{cases}$$

Informally, we say that $\hat{X}$ and $Y$ match such that $Y$ starts at the first position of $\hat{X}$ if the position is solid or as a suffix of one of its non-empty strings if it is elastic-degenerate; and $Y$ ends at the last position of $\hat{X}$ if the position is solid or as a prefix of one of its non-empty strings if it is elastic-degenerate.

**Example 4.2.** *Consider $\hat{X}$ as given in Example 4.1. For string $Y =$* abbcabccacbabb *we have that $\hat{X} \simeq Y$, whereas for string $Z =$* abbccccca*, $\hat{X} \neq Z$*

**Definition 4.8** (Occurrence)**.** In an elastic-degenerate string (text) $\hat{T}$, a solid string (pattern) $P$ is said to have an occurrence starting and ending at positions $i$ and $j$ respectively, if $P \simeq \hat{T}[i..j]$. An occurrence is represented as the pair of starting position $i$ (*head*) and ending position $j$ (*tail*).

For consistency with the intuitive meaning of an occurrence, we say that $P$ occurs at the position of some elastic-degenerate symbol (say $\xi_i$) of $\hat{T}$ if it is a factor of any of the constituent strings of $\xi_i$ (i.e. the starting position and the ending position of that occurrence are the same).

**Example 4.3.** *Consider a pattern $P =$* cabbcb *and a text $\hat{T}$ as follows:*

$$\texttt{aacabbcbbc}\begin{bmatrix}\texttt{a}\\ \texttt{cab}\\ \texttt{acca}\end{bmatrix}\texttt{bb}\begin{bmatrix}\texttt{c}\\ \texttt{acabbcbb}\\ \texttt{cba}\end{bmatrix}\texttt{bacabbc}\begin{bmatrix}\texttt{b}\\ \texttt{cabb}\\ \texttt{bbc}\\ \texttt{aacabb}\end{bmatrix}\texttt{cbc}.$$

*All the occurrences of $P$ in $\hat{T}$ are shown below.*

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| a | a | c | a | b | b | c | b | b | c | $\begin{bmatrix}\texttt{a}\\ \texttt{aab}\\ \texttt{acca}\end{bmatrix}$ | b | b | $\begin{bmatrix}\texttt{c}\\ \texttt{acabbcbb}\\ \texttt{cba}\end{bmatrix}$ | b | a | c | a | b | b | c | $\begin{bmatrix}\texttt{b}\\ \texttt{cabb}\\ \texttt{bbc}\\ \texttt{aacabb}\end{bmatrix}$ | c | b | c |

| **Occurrence:** | (3,8) | (10,14) | (10,15) | (11,14) | (11,15) | (14,14) | (17,22) | (22,24) |
|---|---|---|---|---|---|---|---|---|
| **Strings chosen:** | - | $\xi_1$: a<br>$\xi_2$: cba | $\xi_1$: a<br>$\xi_2$: c | $\xi_1$: acca<br>$\xi_2$: cba | $\xi_1$: acca<br>$\xi_2$: c | $\xi_2$: acabbcbb | $\xi_3$: b<br>or $\xi_3$: bbc | $\xi_3$: cabb<br>or $\xi_3$: aacabb |

Note that more than one occurrence of $P$ can start at the same starting position but their ending positions are different: for instance, $(11,14)$ and $(11,15)$ in Example 4.3. Further note that different strings in the same elastic-degenerate symbols can lead to the same occurrence: for instance, the same pair of head and tail is obtained for occurrences $(17,22)$ and $(22,24)$ in Example 4.3.

**Example 4.4.** *Here, we illustrate the case, where an elastic-degenerate string has the empty string as a seed. The pattern $P =$* babbcb *has an occurrence at* $(2,4)$ *in the text $\hat{T}$ given below:*

$$\hat{T} = \texttt{ab}\begin{bmatrix}\texttt{bcab}\\ \texttt{abb}\end{bmatrix}\begin{bmatrix}\texttt{ab}\\ \texttt{cbb}\\ \texttt{abc}\end{bmatrix}\texttt{cca}\begin{bmatrix}\texttt{bb}\\ \texttt{cb}\end{bmatrix}\texttt{ca}.$$

Here, we formally define the problem to extend the classical pattern matching problem in the context of elastic-degenerate strings.

| PATTERN MATCHING IN ELASTIC-DEGENERATE TEXTS |
| --- |
| **Input:** An elastic-degenerate text $\hat{T} = S_1\xi_1S_2\ldots\xi_{k-1}S_k$ of length $n$ and total size $N$, a pattern $P$ of length $m < N$. |
| **Output:** All the occurrences of $P$ in $\hat{T}$. |

## 4.3 Our Algorithm

By definition, all the occurrences of the pattern $P$ in the text $\hat{T}$ fall under one of the following cases:

1. $P$ entirely lies in some seed.
2. $P$ entirely lies in some string of an elastic-degenerate symbol.
3. $P$ spans across one or more elastic-degenerate symbols. This can further be divided into:
    a) $P$ starts in some seed.
    b) $P$ starts in some string of an elastic-degenerate symbol.

For instance, consider Example 4.3: the occurrences $(3,8)$ and $(14,14)$ fall under Case 1 and Case 2, respectively; $(10,14)$, $(10,15)$, and $(17,22)$ fall under Case 3($a$); $(11,14)$, $(11,15)$, and $(22,24)$ fall under Case 3($b$).

Note that a straightforward solution to this problem would be to find the pattern occurrences in the possibility-set $\Re$ of $\hat{T}$ using the KMP algorithm (see subsection 2.2.1); the running time would be exponential in the number of elastic-degenerate symbols. In this section, we present an efficient algorithm that makes use of the KMP algorithm and the suffix tree data structure. Clearly, the KMP algorithm can easily report the occurrences corresponding to Case 1 and Case 2. Case 3 requires some additional processing and data structures. Our algorithm works in two stages, outlined below.

### Stage 1: Pre-processing

Pre-process the pattern $P$ to compute its failure function as required by the KMP algorithm. In addition, create the generalised suffix tree (see Subsection 2.3.1) $\mathscr{S}_S$ for

the set of strings $\{P, S_1, S_2, \ldots, S_k\}$ corresponding to all the seeds of $\hat{T}$, as well as the generalised suffix tree $\mathscr{S}_\xi$ for the set of strings $\{P\} \cup \xi_1 \cup \xi_2 \cup \ldots \cup \xi_{k-1}$ corresponding to all the strings in each of the elastic-degenerate symbols of $\hat{T}$. Furthermore, preprocess these two suffix trees so as to answer LCP (longest common prefix, explained in Section 2.3) queries in constant time.

## Stage 2: Search

Start searching for the pattern $P$ in the text $\hat{T}$ using the KMP algorithm, comparing the symbols and using the failure function to shift the pattern on a mismatch. The starting position of an occurrence being tested may be either solid or elastic-degenerate; we call the two types of occurrences as *Type* 1 and *Type* 2, respectively. We consider the two types separately as follows:

### Type 1: Solid starting position

Consider a situation where an occurrence starting from a position (say pos) that lies in some seed $S_i$ is being tested. Proceed normally comparing the corresponding symbols of $P$ and $S_i$ and shifting the pattern using the failure function on a mismatch. As soon as the elastic-degenerate symbol $\xi_i$ is encountered (suppose corresponding position in the pattern is p), abort the KMP algorithm (for this test). Check each of the strings of $\xi_i$ (i.e. $E_{i,j}$) for whether or not it occurs in the pattern at position p using LCP queries on $\mathscr{S}_\xi$, and *tick* (mark) the tails of the found occurrences. This can be realised by maintaining a list of (marked/ticked) positions which we denote by $\mathcal{T}_i$.

Next, the subroutine EXTEND (given formally as Subroutine 4.1) is executed. It tries to extend each ticked position of $\mathcal{T}_i$ by testing whether $S_{i+1}$ occurs adjacent to it (using LCP queries on $\mathscr{S}_S$). For each such found occurrence of $S_{i+1}$, occurrences of strings of $\xi_{i+1}$ are checked using the suffix tree $\mathscr{S}_\xi$ and their tails are ticked in $\mathcal{T}_{i+1}$. The procedure will then be repeated for $\mathcal{T}_{i+1}$ and this continues recursively until there is no tail marked in some call. This subroutine also keeps a check on whether $P$ is exhausted in order to report its corresponding occurrence. It is to be noted that an occurrence of $P$ is implied if the length returned by the LCP query between the pattern starting next to some ticked-tail $t$ and either of the following hits the boundary (end) of the pattern:

- some seed $S_i$

- any string $E_{i,j}$ of some elastic-degenerate symbol $\xi_i$.

Once the subroutine ends (reporting all the occurrences of $P$ starting from `pos`, if any), the failure function corresponding to the position where the KMP algorithm was aborted (i.e. $p$) is used to shift the pattern, and the KMP algorithm resumes. Figure 4.1 and Table 4.1 abstractly elucidates the description given above.

**Subroutine 4.1** EXTEND : Extends ticked tails in a given $\mathcal{T}_i$ and reports the occurrences found, if any.

```
1: function EXTEND(T_i)
2:     isNonEmpty ← false                                   ▷ A flag
3:     for all t ∈ T_i do
4:         ℓ_s ← |LCP(P[t+1..m], S_{i+1}[1..|S_{i+1}|])|
5:         if (ℓ_s + t) = m then                            ▷ Pattern ends
6:             Report the occurrence
7:         else if ℓ_s = |S_{i+1}| then                     ▷ S_{i+1} occurs here
8:             e ← t + |S_{i+1}|;
9:             for all E_{i+1,j} ∈ ξ_{i+1} do
10:                ℓ_e ← |LCP(P[e+1..m], E_{i+1,j}[1..|E_{i+1,j}|])|
11:                if (ℓ_e + e) = m then                    ▷ Pattern ends
12:                    Report the occurrence (if not reported already)
13:                else if ℓ_e = |E_{i+1,j}| then           ▷ E_{i+1,j} occurs here
14:                    Mark e + |E_{i+1,j}| in T_{i+1}
15:                    isNonEmpty ← true
16:                end if
17:            end for
18:        end if
19:    end for
20:    if isNonEmpty then
21:        EXTEND(T_{i+1});
22:    end if
23: end function
```

### Type 2: Elastic-Degenerate starting position

Consider a situation where the starting position of an occurrence to be tested is an elastic-degenerate symbol $\xi_i$. This case can be processed in a similar fashion as the one described for Type 1, with the only difference being the manner in which tails are ticked initially.

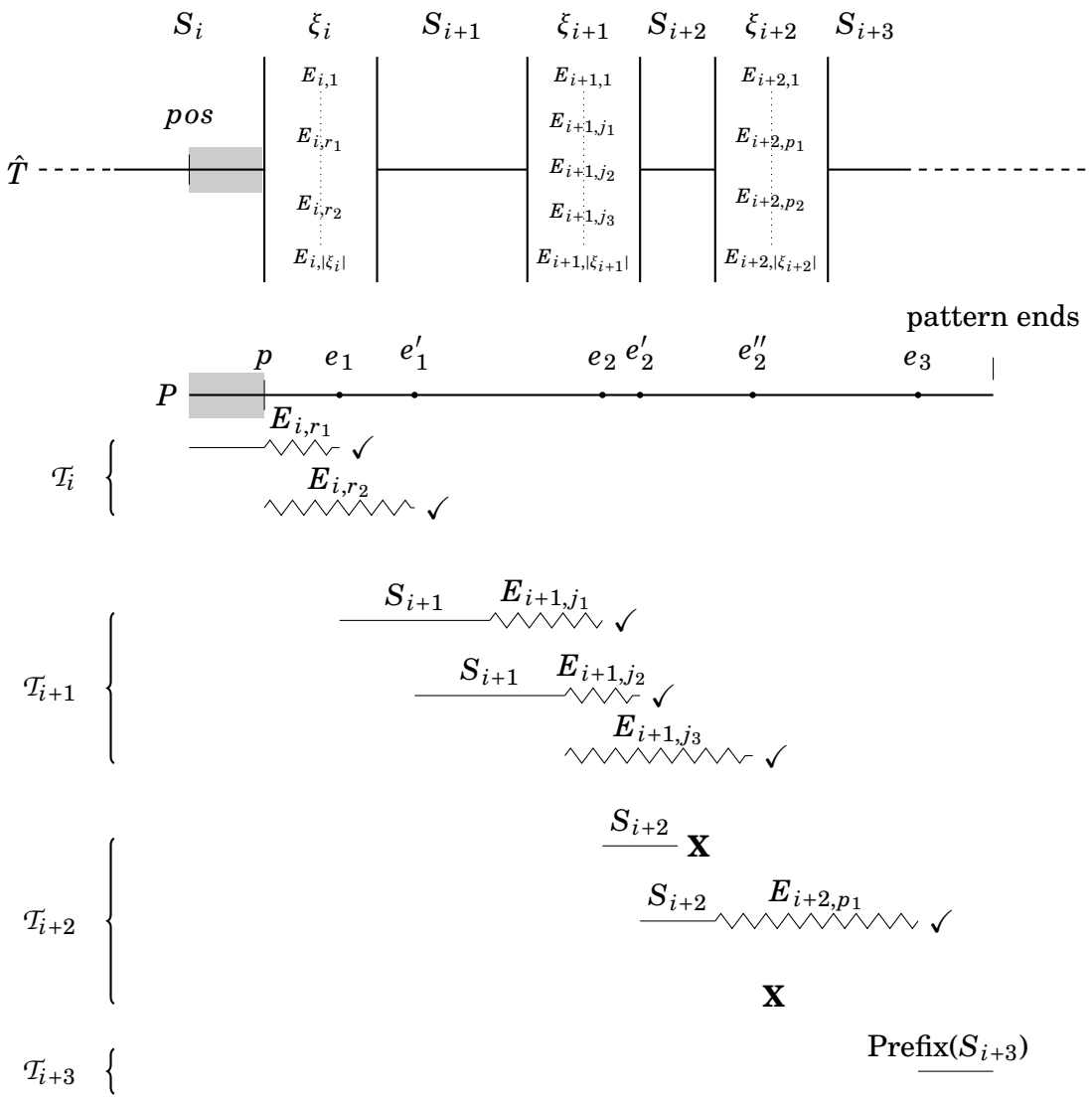


Figure 4.1: An illustration of how the algorithm works for Type 1 occurrences. Strings in elastic-degenerate symbols are shown as zigzag, while solid lines depict the seeds. The grey area represents the initial (solid) match. Symbol **X** denotes that this path could not be extended further while the symbol ✓ represents a ticked tail.

Begin by applying the KMP algorithm for each $E_{i,j}$ to achieve two purposes: finding the occurrences of $P$ in $E_{i,j}$ and ticking the last positions of suffixes of $E_{i,j}$ that appear as prefixes of $P$. The ticked tails obtained in that way are then extended by the subroutine EXTEND recursively and occurrences are reported. After the subroutine EXTEND ends, the KMP algorithm resumes and the testing starts at the beginning of the seed $S_{i+1}$.

| | | | |
|---|---|---|---|
| Initial | | | |
| $p$: | | | |
| | | $E_{i,r_1}$ next to $p$ | Tick $e_1$ |
| | | $E_{i,r_2}$ next to $p$ | Tick $e'_1$ |
| | $\mathcal{T}_i = [e_1, e'_1]$ | | |
| Iteration 1 | | | |
| $e_1$: | $S_{i+1}$ next to $e_1$ | | |
| | | $E_{i,j_1}$ follows this occurrence | Tick $e_2$ |
| $e'_1$: | $S_{i+1}$ next to $e'_1$ | | |
| | | $E_{i+1,j_2}$ follows this occurrence | Tick $e'_2$ |
| | | $E_{i+1,j_3}$ follows this occurrence | Tick $e''_2$ |
| | $\mathcal{T}_{i+1} = [e_2, e'_2, e''_2]$ | | |
| Iteration 2 | | | |
| $e_2$: | $S_{i+2}$ next to $e_2$ | | |
| | | No string from $\xi_{i+2}$ follows this occurrence | No Extension |
| $e'_2$: | $S_{i+2}$ next to $e'_2$ | | |
| | | $E_{i+2,p_1}$ follows this occurrence | Tick $e_3$ |
| $e''_2$: | No $S_{i+2}$ next to $e''_2$ | | No extension |
| | $\mathcal{T}_{i+2} = [e_3]$ | | |
| Iteration 3 | | | |
| $e_3$: | Prefix of $S_{i+3}$ next to $e_3$ | | |
| | | Pattern exhausted | Report occurrence |
| Nothing to extend | Exit the procedure | | |

Table 4.1: Table representing the progress of Subroutine 4.1 for the abstraction shown in Fig 4.1.

## 4.4 Analysis of the Algorithm

In this section, we discuss the correctness of the algorithm and analyse its space and time complexity.

### 4.4.1 Correctness

Consider an occurrence $(i, j)$. If the occurrence falls under Case 1 (resp. Case 2) then $j = i + m - 1$ (resp. $j = i$) for some fixed $i$. Thus, the number of occurrences falling under either Case 1 or Case 2 is bounded by $\mathcal{O}(n)$. On the other hand, for occurrences under Case 3, let the parameter $\gamma$ represent the maximum number of elastic-degenerate symbols that any occurrence $(i, j)$ may span. Note that $\gamma$ captures the possibility that the elastic-degenerate symbols contain empty strings. As there can be maximum $m$ prefixes going past an elastic-degenerate position, the number of occurrences per starting position $i$ are bounded by $\mathcal{O}(\gamma m)$. Thus the total number of distinct occurrences $(i, j)$ is bounded by $\mathcal{O}(\gamma mn)$.

The correctness of the presented algorithm is straightforward as every starting position of the text is being tested for potential occurrences exhaustively. While the occurrences corresponding to Case 1 and Case 3($a$) are covered by Type 1, Type 2 investigates every occurrence associated with Case 2 and Case 3($b$). Thus, all the occurrences of $P$ in $\hat{T}$ are reported.

### 4.4.2 Space Complexity

The space required by both, the failure function and ticked tails list, is $\mathcal{O}(m)$. The suffix tree $\mathcal{S}_S$ uses $\mathcal{O}(m + \sum_{i=1}^{k} |S_i|)$ space and the suffix tree $\mathcal{S}_\xi$ uses $\mathcal{O}(m + \sum_{i=1}^{k-1} \sum_{j=1}^{|\xi_i|} |E_{i,j}|)$ space. This leads to the total space required to be $\mathcal{O}(N)$, as $\sum_{i=1}^{k} |S_i| + \sum_{i=1}^{k-1} \sum_{j=1}^{|\xi_i|} |E_{i,j}| = N$ and $m < N$.

### 4.4.3 Time Complexity

The time taken by the pre-processing stage is $\mathcal{O}(N)$ as the failure function can be computed in $\mathcal{O}(m)$ time and construction of both the suffix trees (along with their pre-processing required to answer LCP queries in constant time) can be done in $\mathcal{O}(N)$ time.

The search stage uses the KMP algorithm over each seed and each string of every elastic-degenerate symbol in the text to report the occurrences for Case 1 and Case 2, and to search the beginning of the occurrence for Case 3. Thus the time consumed by the KMP algorithm is $\mathcal{O}(\sum_{i=1}^{k} |S_i| + \sum_{i=1}^{k-1} \sum_{j=1}^{|\xi_i|} |E_{i,j}|) = \mathcal{O}(N)$.

The subroutine EXTEND can be analysed as follows. Intuitively, for every ticked position in the pattern (which can at most be $m$), an LCP query is used to find whether the succeeding seed occurs at the ticked position or not. Such an occurrence is then tried to be extended by another LCP query with each of the strings in the following elastic-degenerate symbol. Let parameter $\alpha$ represent the maximum number of strings in any elastic-degenerate symbol of the text. This extension step for each ticked position will be carried out at most $\alpha$ times. More specifically, the outer loop of the subroutine runs $m$ times and the inner one takes $\mathcal{O}(\alpha)$ time, as each LCP query takes constant time. Thus, each recursive call requires $\mathcal{O}(m\alpha)$ time. The number of recursive calls depends on the number of elastic-degenerate symbols spanned by the longest occurrence of any prefix of $P$ starting at the position being tested. In other words, if an occurrence of the longest prefix spans across $i$ elastic-degenerate symbols, there will be $i$ recursive calls to the procedure. If a parameter $\gamma$ were to reflect the maximum such $i$ in an occurrence of $P$ then EXTEND could be executed in $\mathcal{O}(\alpha\gamma m)$ time in total for each starting position. It would require an additional check (at Line 20) whether $i < \gamma$ before making a recursive call to EXTEND. Note that $\gamma$ is a user-defined parameter that is upper-bounded by $k$ i.e. the number of elastic-degenerate symbols. In practice, $\gamma = cm$ for a small constant $c$ should make a more *sensible* choice.

Initial ticking of the tails in Type 1 needs $\mathcal{O}(\alpha)$ time. For Type 2, initial ticking is done by the KMP algorithm (already accounted for above). In the worst case, EXTEND will be called from each of the $n$ starting positions of the text, leading to an overall time-complexity of the algorithm to be $\mathcal{O}(N + \alpha\gamma mn)$. In other words, the algorithm takes $\mathcal{O}(N + \alpha\gamma mn)$ time to find and report $\mathcal{O}(\gamma mn)$ number of possible occurrences of the pattern.

## 4.5 Experimental Results

A proof of concept implementation of the algorithm was developed (in C++). The implementation is openly available on GitHub[i], along with the synthetic data used in the following experiments. The tool was compiled with g++ version 4.7.3 at optimisation level 3 (-O3). The following experiments were conducted on a desktop computer using one core of Intel® Core™ i7-2600S CPU at 2.8GHz and 8GB of RAM under 64-bit GNU/Linux

The experiments were set up to verify the accuracy and corroborate the asymptotic behaviour of the algorithm by studying its performance on synthetic data similar to the datasets used in genomics. Data was generated using random uniform distribution for an alphabet of size 4 (which is same as the alphabet size in genomic data) – namely, `A, C, G, T`.

### Practical details

Note that the current implementation of the algorithm uses the *enhanced suffix array* (Subsection 2.3.2) for answering LCP queries in constant time (after linear time pre-processing) instead of the suffix tree data structure; this is because of its space-efficiency, although it is easier to explain the algorithm using a suffix tree. In addition, ticked positions are being maintained as a boolean array (rather than a list) so that the occurrences at a specific starting position can be reported in an ordered fashion which makes the verification of results quicker. These practical modifications do not influence the theoretical bounds in the complexity analysis.

### 4.5.1 Accuracy

To test the accuracy, our aim was to test whether or not our algorithm could report all (and only) the positions of occurrences of the pattern $P$ in the $\hat{T}$. It was validated using carefully designed data – a random text sequence of specific length was first generated using 3 letters of the alphabet; a pattern of specified length containing a single occurrence of the fourth letter was manually designed; the pattern was inserted in the text sequence at several places, thus ensuring different types of occurrences (entirely in one seed, entirely in one or more strings of some elastic-

[i] https://github.com/Ritu-Kundu/ElDeS

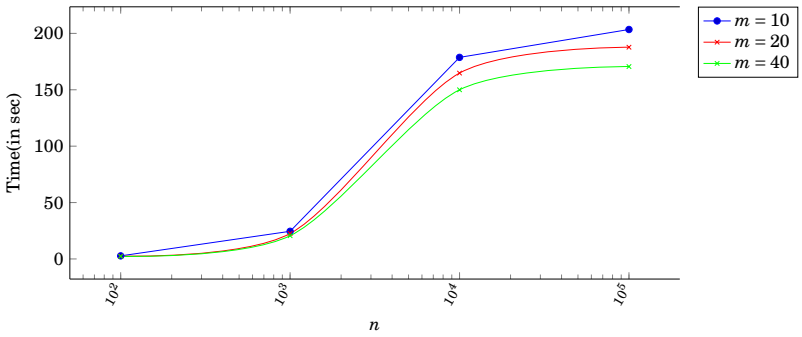


Figure 4.2: Plot showing the running time vs the text-length $n$ for various values of $m$.

degenerate symbol, spanning across several elastic-degenerate symbols) and these insertion-positions were recorded. The reported occurrences were then verified against the recorded positions. In each of such runs, the algorithm successfully passed the test, correctly reporting all the indices where the pattern occurred (without missing any occurrence or reporting any extraneous occurrence).

### 4.5.2 Performance

To study performance, text sequences of exponentially varying lengths were considered. The number of degenerate symbols was set to 10% of the text length ($n$). The length of a string within a symbol was upper-bounded by 10. The number of strings within a symbol ($\alpha$) was chosen randomly with an upper-bound of 10. Each text so generated was used to find occurrences of randomly-generated patterns with varying pattern lengths ($m = 10, 20, 40$). Ten such sets were repeated and the average of the running-times was recorded. Figure 4.2 presents the graphs showing the average time taken by the algorithm to run versus the length of the text $n$. Note that as $m$ increases, the running-time decreases for the same $n$. This is because of the reduced number of occurrences as a result of the decreased probability of finding a random pattern in a random text with an increase in pattern length.

## 4.6 Impact

Our proposed model – elastic-degenerate strings – for capturing the uncertainty that arises when a single representation is used for a collection of similar textual sequences, has successfully attracted the attention of researchers working in the

field of Stringology. The interest of the research community can be gauged by a spate of the publications that started emerging immediately after we introduced this model. Several improved algorithms have been proposed since then, albeit after modifying the definition of an occurrence. While our algorithm, presented in this chapter, reports the starting and ending positions of an occurrence of the pattern, all the subsequent algorithms report only the ending position of an occurrence. Nevertheless, considerable improvement in the time-complexity of the solution to the pattern matching problem in this setting has been achieved – Table 4.2 chronologically presents the algorithms that have been proposed in the short time-span since its introduction to the time of writing this dissertation.

Table 4.2: Subsequent Algorithms on Elastic-Degenerate String Matching Problem

| Variant | Publication | Time Complexity | |
|---|---|---|---|
| Online | Grossi et al. [GIL$^+$17] | Algorithm 1 : $\mathcal{O}(nm^2+N)$ after $\mathcal{O}(m\lceil\frac{m}{w}\rceil)$ preprocessing | $w$ is computer word length. |
| | | Algorithm 2 : $\mathcal{O}(N\lceil\frac{m}{w}\rceil)$ after $\mathcal{O}(m\lceil\frac{m}{w}\rceil)$ preprocessing | |
| With errors | Bernardini et al. [BPPR17] | Algorithm 1 : $\mathcal{O}((k+1)^2mG+(k+1)N)$ with insertions / deletions / substitutions | $k$ is the number of errors allowed. |
| | | Algorithm 2 : $\mathcal{O}((k+1)(mG+N))$ with substitutions only | $G$ is the number of all the substrings constituting seeds and elastic-degenerate symbols. |
| Online | Aoyama et al. [ANI$^+$18] | $\mathcal{O}(nm\sqrt{m\log m}+N)$ | |
| Multiple Patterns (Online) | Pissis and Retha [PR18] | $\mathcal{O}(N\lceil\frac{M}{w}\rceil)$-time with preprocessing time $O(M)$ | $M$ is the total length of the patterns. |
| Online | Cisłak et al. [CGH18] | $\mathcal{O}(N\lceil\frac{m}{w}\rceil)$-time | Practical improvement of an order of magnitude and alphabet independence. |

CHAPTER 5

# LONGEST UNBORDERED FACTOR ARRAY

A *border* $u$ of a word $w$ is a proper factor of $w$ occurring both as a prefix and as a suffix. The *maximal unbordered factor* of $w$ is the longest factor of $w$ which does not have a border. Here, an $\mathcal{O}(n \log n)$-time (with high probability) or $\mathcal{O}(n \log n \log^2 \log n)$-time (deterministic) algorithm to compute the *Longest Unbordered Factor Array* of $w$ for general alphabets is presented, where $n$ is the length of $w$. This array specifies the length of the maximal unbordered factor starting at each position of $w$. This is a major improvement on the running time of the previously best worst-case algorithm working in $\mathcal{O}(n^{1.5})$ time for integer alphabets [Gawrychowski et al., 2015].

This chapter is organised as follows: in the first section, we summarise the associated results in literature and present the previous state-of-the-art algorithm. In Section 5.2, we present the preliminaries, a formal definition of the problem, and some useful properties of unbordered words. We lay down the combinatorial foundation of the algorithm in Section 5.3 and expound the algorithm in Section 5.4. The analysis of algorithm has been explicated in Section 5.5. Lastly, in Section 5.6, we provide an observation that can further accelerate the algorithm in practice.

## 5.1 Background

There are two central properties characterising repetitions in a word – *period* and *border* – which play direct or indirect roles in several diverse applications ranging over pattern matching, text compression, assembly of genomic sequences and so

on (see [CHL07, CR02]). A period of a non-empty word $w$ of length $n$ is an integer $p$, $1 \leq p \leq n$, such that $w[i] = w[i+p]$, for all $i$, $1 \leq i \leq n-p$. For instance, 3, 6, 7, and 8 are periods of the word aabaabaa. On the other hand, a border $u$ of $w$ is a (possibly empty) proper factor of $w$ occurring both as a prefix and as a suffix of $w$. For example, $\varepsilon$, a, aa, and aabaa are the borders of $w$ = aabaabaa.

In fact, the notions of border and period are dual: the length of each border of $w$ is equal to the length of $w$ minus the length of some period of $w$. For example, aa is a border of the word aabaabaa; it corresponds to period $6 = |\texttt{aabaabaa}| - |\texttt{aa}|$. Consequently, the basic data structure of periodicity on words is the *border array* which stores the length of the longest border for each prefix of $w$ (border table/array has been introduced in Section 2.1). The computation of the border array of $w$ was the fundamental concept behind the first linear-time pattern matching algorithm – given a word $w$ (pattern), find all its occurrences in a longer word $y$ (text). The border array of $w$ is better known as the "failure function" introduced in [MJP70] (see also [AHU87]). It is well-known that the border array of $w$ can be computed in $\mathcal{O}(n)$ time, where $n$ is the length of $w$, by a variant of the Knuth-Morris-Pratt algorithm [MJP70].

Another notable aspect of the inter-dependency of these dual notions is the relationship between the length of the maximal unbordered factor of $w$ and the periodicity of $w$. A maximal unbordered factor is the longest factor of $w$ which does not have a border; its length is usually represented by $\mu(w)$, e.g. the maximal unbordered factor is aabab and $\mu(w) = 5$ for the word $w$ = baabab. This dependency has been a subject of interest in the literature for a long time, starting from the 1979 paper of Ehrenfeucht and Silberger [ES79] in which they raised the question – at what length of $w$, expressed in terms of $\mu(w)$, is $\mu(w)$ maximal (i.e. equal to the minimal period of the word as it is well-known that it cannot be longer than that). This line of questioning, after being explored for more than three decades, culminated in 2012 with the work by Holub and Nowotka [HN12] where an asymptotically optimal upper bound ($\mu(w) \leq \frac{3}{7}n$) was presented; a historic overview of the related research can be found in [HN12].

Somewhat surprisingly, the symmetric computational problem—given a word $w$, compute the longest factor of $w$ that does not have a border—had not been studied until very recently. In 2015, Kucherov et al. [KLS15] considered this arguably natural problem and presented the first sub-quadratic-time solution. A naïve way to solve this problem is to compute the border array starting at each position of $w$

and locating the rightmost zero, which results in an algorithm with $\mathcal{O}(n^2)$ worst-case running time. On the other hand, the computation of the longest unbordered factor can be done in linear time for the cases when $\mu(w)$ or its minimal period is small (i.e. at most half the length of $w$) using the linear-time computation of unbordered conjugates [DLL14]. However, as has been illustrated in [KLS15] and [CK16], most of the words do not fall in this category owing to the fact that they have large $\mu(w)$ and consequently large minimal period; more specifically, the expected length of the maximal unbordered factor of a word $w$ of length $n$ over an alphabet of size $\sigma$ has been shown to be at least $0.99n$ (for sufficiently large $n$ and $\sigma > 4$) and $n - \mathcal{O}(\sigma^{-1})$, respectively. In [KLS15], an adaptation of the basic algorithm (that uses the border array) has been provided with average-case running time $\mathcal{O}(n + n^2/\sigma^4)$, where $\sigma$ is the alphabet's size; it has also been shown to work better, both in practice and asymptotically, than another straightforward approach that employs the data structures from [KRRW15, KRRW12] to query all relevant factors.

### 5.1.1 Previously Best Algorithm

The previously best worst-case algorithm to compute the maximal unbordered factor of a given word takes $\mathcal{O}(n^{1.5})$ time. It was presented by Gawrychowski et al. [GKSS15] and it works for integer alphabets (alphabets of polynomial size in $n$). This algorithm works by categorising bordered factors into those having *short* borders and those having *long* borders depending on a threshold, and exploiting the facts that the short borders for each position are bounded by the threshold and the factors with only long borders are small in number. More precisely, a border is considered short or long depending on whether it is shorter or longer than the threshold which is set to $\sqrt{n}$; any unbordered substring of length $\leq 4\sqrt{n}$ can be found naïvely by utilising the border table approach. For computing the unbordered substrings of longer lengths, the algorithm conceptually divides $w$ into blocks of length $\sqrt{n}$ and progresses in $\sqrt{n}$ stages. In each stage (say $k$), the algorithm finds a set of substrings ($\mathbb{F}_k^i$) starting at some position $i$ (which may result in a substring longer than $4\sqrt{n}$) and ending in the $k^{th}$ block (interval $[(k\sqrt{n}+1,(k+1)\sqrt{n}]$). This scanning is followed by *carefully* selecting a *candidate* from $\mathbb{F}_k^i$ such that the candidate does not have a short border and if it is unbordered, it is the longest unbordered string in the set. In turn, the candidate is tested for whether it is unbordered – if it is unbordered and longer than the longest unbordered factor found so far, this candidate starts reflecting the longest unbordered factor so far.

Gawrychowski et al. [GKSS15] also presented another algorithm that runs in $\mathcal{O}(n \log n)$ time on average and $\mathcal{O}(n^2)$ time in the worst case. More recently, an $\mathcal{O}(n)$-time average-case algorithm was presented using the straightforward border table approach and exploiting a refined bound on the expected length of the maximal unbordered factor [CK16].

**Our Contribution**. In this chapter, we show how to efficiently answer the Longest Unbordered Factor question using combinatorial insights[i]. Specifically, we present an algorithm that computes the *Longest Unbordered Factor Array* in $\mathcal{O}(n \log n)$ time with high probability. The algorithm can also be implemented deterministically in $\mathcal{O}(n \log n \log^2 \log n)$ time. This array specifies the length of the maximal unbordered factor at each position in $w$. We thus improve on the running time of the currently fastest algorithm, which reports only the maximal unbordered factor of $w$ and works only for integer alphabets, taking $\mathcal{O}(n^{1.5})$ time. Moreover, we show that the analysis of our algorithm is tight: an infinite family of words that exhibit the worst-case behaviour of the algorithm is provided.

## 5.2 Preliminaries

Throughout this chapter, we consider a non-empty word $w$ of length $n$ over a *general alphabet* $\Sigma$; in this case, we replace each letter by its rank such that the resulting word consists of integers in the range $[1, \ldots, n]$. This can be done in $\mathcal{O}(n \log n)$ time after sorting the letters of $\Sigma$. We begin by recollecting and expanding the premises defining period and border in Chapter 2.

An integer $p$, $1 \leq p \leq n$ is a *period* of $w$ if and only if $w[i] = w[i + p]$ for all $i$, $1 \leq i \leq n - p$. The smallest period of $w$ is called the **minimum period** (or **the period**) of $w$. A word $u$ is a *border* of $w$, if $w = uv = v'u$ for some non-empty words $v$ and $v'$; note that $u$ is both a proper prefix and a proper suffix of $w$. It should be clear that if $w$ has a border of length $|w| - p$ then it has a period $p$. Thus, the minimum period of $w$ corresponds to the length of the **longest border** (or **the border**) of $w$. Observe that the empty word $\varepsilon$ is a border of any word $w$. If $u$ is the **shortest border** then $u$ is the shortest *non-empty* border of $w$.

The word $w$ is called **bordered** if it has a non-empty border, otherwise it is **unbordered**. Equivalently, the minimum period $p = |w|$ for an unbordered word $w$.

[i]The corresponding software-tool is available on GitHub at https://github.com/Ritu-Kundu/luf.

Note that every bordered word $w$ has a shortest border $u$ ($u \neq w$ and $u \neq \varepsilon$) such that $w = uvu$, where $u$ is unbordered. By $\mu(w)$ we denote the maximum length among all the unbordered factors of $w$.

The LONGEST UNBORDERED FACTOR ARRAY problem is defined as follows.

| LONGEST UNBORDERED FACTOR ARRAY |
|---|
| **Input:** A word $w$ of length $n$. |
| **Output:** An array LUF$[1..n]$ such that LUF$[i]$ is the length of the longest unbordered factor starting at position $i$ in $w$, for all $1 \leq i \leq n$. |

**Example 5.1.** *Consider* $w =$ aabbabaabbaababbabab. *The longest unbordered factor array is as follows. (Observe that* $w$ *is unbordered thus* $\mu(w) = |w| = 20$.*)*

| $i$ | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $w[i]$ | a | a | b | b | a | b | a | a | b | b | a | a | b | a | b | b | a | b | a | b |
| LUF[i] | 20 | 3 | 12 | 9 | 12 | 3 | 14 | 3 | 11 | 3 | 10 | 5 | 2 | 3 | 5 | 2 | 2 | 2 | 2 | 1 |

### 5.2.1 Useful Properties of Unbordered Words.

Recall that a word $u$ is a border of a word $w$ if and only if $u$ is both a proper prefix and a suffix of $w$. A border of a border of $w$ is also a border of $w$. A word $w$ is unbordered if and only if it has no non-empty border; equivalently $\varepsilon$ is the only border of $w$. The following properties related to unbordered words form the basis of our algorithm and were presented and proved in [Duv82].

**Proposition 5.1** ([Duv82])**.** *Let* $w$ *be a bordered word and* $u$ *be the shortest non-empty border of* $w$*. The following propositions hold:*

1. $u$ *is an unbordered word;*
2. $u$ *is the unique unbordered prefix and suffix of* $w$*;*
3. $w$ *has the form* $w = uvu$*.*

**Proposition 5.2** ([Duv82])**.** *For any word* $w$*, there exists a unique sequence* $\langle u_1, \cdots, u_k \rangle$ *of unbordered prefixes of* $w$ *such that* $w = u_k \cdots u_1$*. Furthermore, the following properties hold:*

1. $u_1$ *is the shortest border of* $w$*;*

2. *$u_k$ is the longest unbordered prefix of $w$;*

3. *for all $i$, $1 \le i \le k$, $u_i$ is an unbordered prefix of $u_k$.*

The computation of the unique sequence described in Proposition 5.2 provides a unique **unbordered-decomposition** of a word. For instance, for $w = \texttt{baababbabab}$, the unique unbordered-decomposition of $w$ is $\texttt{baa}\cdot\texttt{ba}\cdot\texttt{b}\cdot\texttt{ba}\cdot\texttt{ba}\cdot\texttt{b}$.

## 5.3 Computational Tools

In what follows, we introduce a data structure and present some combinatorial properties that will be used by our algorithm as computational tools.

### 5.3.1 Longest Successor Factor (Length and Reference) Arrays

The longest successor factor of $w$ (denoted by *lsf*) starting at position $i$, is the longest factor of $w$ that occurs at $i$ and has at least one other occurrence in the suffix $w[i+1..n]$. The **longest successor factor array** ($\mathsf{LSF}_\ell$) gives for each position $i$ in $w$, the length of the longest factor starting both at position $i$ and at another position $j > i$. Formally, the longest successor factor array ($\mathsf{LSF}_\ell$) is defined as follows.

$$\mathsf{LSF}_\ell[i] = \begin{cases} 0 & \text{if } \ i = n, \\ \max\{k \mid w[i..i+k-1] = w[j..j+k-1\}, & \text{for } \ i < j \le n. \end{cases}$$

Additionally, we define the **LSF-Reference Array**, denoted by $\mathsf{LSF}_r$. This array specifies, for each position $i$ of $w$, the *reference* of the longest successor factor at $i$. The **reference** of $i$ is defined as the position $j$ of the last occurrence of $w[i..i+\mathsf{LSF}_\ell[i]-1]$ in $w$; we say $i$ **refers to** $j$. Formally, LSF-Reference Array ($\mathsf{LSF}_r$) is defined as follows.

$$\mathsf{LSF}_r[i] = \begin{cases} nil & \text{if } \ \mathsf{LSF}_\ell[i] = 0, \\ \max\{j \mid w[j..j+\mathsf{LSF}_\ell[i]-1] = w[i..i+\mathsf{LSF}_\ell[i]-1]\} & \text{for } \ i < j \le n. \end{cases}$$

**Computation:**

Note that the longest successor factor array is a mirror image of the well-studied longest previous factor array which can be computed in $\mathcal{O}(n)$ time for integer alphabets [CI08, CIK$^+$10, CII$^+$13]. Moreover, in [CI08], an additional array that keeps a position of some previous occurrence of the longest previous factor was presented;

such a position may not be the leftmost one. Arrays $\mathsf{LSF}_\ell$ can be computed using simple modifications (pertaining to the symmetry between the longest previous and successor factors) of this algorithm within $\mathcal{O}(n)$ time for integer alphabets. The modified algorithm also computes a position $j > i$ of each factor $w[i\,..\,i+|\mathsf{LSF}_\ell[i]|-1]$, where $1 \le i \le n$. Each such factor corresponds to the lowest common ancestor of the two nodes in the suffix tree of $w$ representing the suffixes $i$ and $j$, which can be identified in constant time (see Subsection 2.3.1). A linear-time pre-processing of the suffix tree allows the computation of the rightmost position of each such factor in constant time, thus yielding the array $\mathsf{LSF}_r$ .

**Example 5.2.** *Let* $w =$ `aabbabaabbaababbabab`. *The associated arrays are as follows.*

| $i$ | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $w[i]$ | a | a | b | b | a | b | a | a | b | b | a | a | b | a | b | b | a | b | a | b |
| $\mathsf{LSF}_\ell[i]$ | 5 | 6 | 5 | 4 | 3 | 4 | 3 | 4 | 3 | 2 | 1 | 4 | 3 | 2 | 1 | 3 | 2 | 1 | 0 | 0 |
| $\mathsf{LSF}_r[i]$ | 7 | 14 | 15 | 16 | 17 | 10 | 11 | 14 | 15 | 18 | 19 | 17 | 18 | 19 | 20 | 18 | 19 | 20 | nil | nil |

**Remark 5.1.** *For brevity, we will use lsf and luf to represent the longest successor factor and the longest unbordered factor, respectively.*

### 5.3.2 Combinatorial Tools

The core of our algorithm exploits the unique unbordered-decomposition of all suffixes of $w$ in order to compute the length of the longest unbordered prefix of each such suffix. Let the unbordered-decomposition of $w[i\,..\,n]$ be $u_k \cdots u_1$ as in Proposition 5.2. Then $\mathsf{LUF}[i] = |u_k|$. In order to compute the unbordered-decomposition for all the suffixes *efficiently*, the algorithm uses the repetitive structure of $w$ characterised by the longest successor factor arrays.

**Basis of the algorithm.** Abstractly, it is easy to observe that for a given position, if the length of the longest successor factor is zero (no factor starting at this position repeats afterwards) then the suffix starting at that position is necessarily unbordered. On the other hand, if the length of the longest successor factor is smaller than the length of the unbordered factor at the reference (the position of the last occurrence of the longest successor factor) then the ending positions of the longest unbordered factors at this position and that at its reference will coincide.

The remaining case is not straightforward and its handling accounts for the bulk of the algorithm. The following lemmas formalise the essence of the algorithm.

**Lemma 5.1.** *If* $\mathsf{LSF}_\ell[i] = 0$ *then* $\mathsf{LUF}[i] = n - i + 1$*, for* $1 \le i \le n$*.*

**Proof.** The statement implies that suffix $w[i..n]$ is unbordered. Assume the contrary, that $w[i..n]$ is bordered, and let $\beta > 0$ be the length of its longest border. Then $w[i..i+\beta-1] = w[n-\beta+1..n]$. Hence $\mathsf{LSF}_\ell[i] \ge \beta$ which is a contradiction. ■

**Lemma 5.2.** *If* $\mathsf{LSF}_r[i] = j$ *and* $\mathsf{LSF}_\ell[i] < \mathsf{LUF}[j]$ *then, for* $1 \le i \le n$*,*

$$\mathsf{LUF}[i] = j + \mathsf{LUF}[j] - i$$

**Proof.** Let $k = j + \mathsf{LUF}[j] - 1$, $u = w[j..k]$ and $v = w[i..i + \mathsf{LSF}_\ell[i] - 1]$; refer to Figure 5.1. We first show that the factor $w[i..k]$ is unbordered. On the contrary, assume that $w[i..k]$ is bordered and let $\beta$ be the length of one of its borders ($\beta < \mathsf{LSF}_\ell[i]$ as $\mathsf{LSF}_r[i] = j$). This implies that $w[i..i+\beta-1] = w[k-\beta+1..k]$. Since $w[j..j + \mathsf{LSF}_\ell[i] - 1] = v$, we get $w[j..j+\beta-1] = w[k-\beta+1..k]$ (i.e. $u$ is bordered) which is a contradiction. Moreover, $w[k+1..n]$ can be factorised into prefixes of $u$ (by definition of $\mathsf{LUF}$); every such prefix is also a proper prefix of $v$ which will make every factor $w[i..k']$, $k < k' \le n$ to be bordered. Therefore, $w[i..k]$ is the longest unbordered factor at $i$. ■

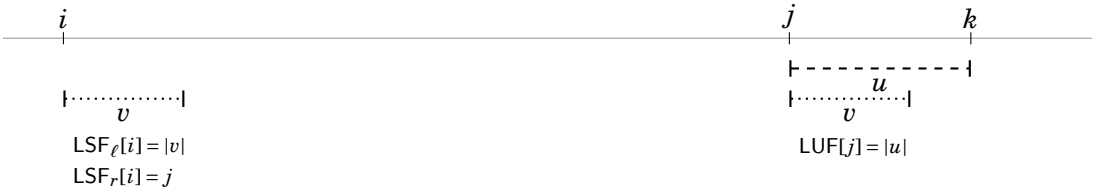


Figure 5.1: Illustration of the case when $i$ refers to $j$ such that $v$ is the *lsf* at $i$, $u$ is the *luf* at $j$ and $|v| < |u|$.

Now, we introduce the notion of the *hook* that aids in finding the unbordered-decomposition of suffixes $w[i..n]$ for the remaining case (i.e. $\mathsf{LSF}_\ell[i] \ge \mathsf{LUF}[\mathsf{LSF}_r[i]]$).

**Definition 5.1** (Hook)**.** The *hook* of a position $j$ given a length $\ell > 0$, denoted by $\mathscr{H}^\ell_j$, is the smallest position $q$ such that the non-empty factor $w[q..j-1]$ can be

decomposed into unbordered prefixes of $w[j..j+\ell-1]$. Formally, $q$ is $\mathscr{H}^{\ell}_{j}$ if $q$ is the smallest position such that

$$w[q..j-1] = u_r \cdot u_{r-1} \cdot \ldots \cdot u_2 \cdot u_1$$

where each $u_i$, $1 \le i \le r$, is an unbordered prefix of $w[j..j+\ell-1]$.

The following observation provides a greedy construction of this decomposition.

**Observation 5.1.** *The decomposition of a word $v$ into unbordered prefixes of another word $u$ is unique. Such a decomposition can be constructed by iteratively trimming the shortest prefix of $u$ which occurs as a suffix of the decomposed word.*

In other words, $\mathscr{H}^{\ell}_{j}$ gives the position such that the factor $w[\mathscr{H}^{\ell}_{j}..j-1]$ is the longest non-empty suffix of $w[1..j-1]$ that can be factorised from right to left into the shortest prefixes of $u$ (the factor of length $\ell$ starting at $j$). The factorisation is done by finding the shortest prefix of $u$ ending at $j-1$ (say $u_1$), then the shortest prefix of $u$ preceding $u_1$ (say $u_2$), and so on. If either $\ell = 0$ or no prefix of $u$ matches a proper suffix of $w[1..j-1]$, then $\mathscr{H}^{\ell}_{j} = j$.

Moreover, the decomposability into unbordered prefixes of $u$ is hereditary in a certain sense:

**Observation 5.2.** *If a word $v$ can be decomposed into unbordered prefixes of $u$, then every prefix of $v$ also admits such a decomposition. Formally, if $v = u_r \cdot u_{r-1} \cdot \ldots \cdot u_2 \cdot u_1$ such that each $u_i$, $r \ge i \ge 1$ is an unbordered prefix of $u$ then any prefix $v[1..k]$ can be uniquely decomposed as*

$$v[1..k] = u_r \cdot u_{r-1} \cdot \ldots \cdot u_{i-1} \cdot u'_p \cdot u'_{p-1} \cdot \ldots \cdot u'_1$$

*where position $k$ falls in $u_i$ and each $u'_j$, $p \ge j \ge 1$ is an unbordered prefix of $u$; simply, the decomposition preceding $u_i$ will be retained by the prefix.*

**Example 5.3.** *Consider* $w =$ `aabbabaabbaababbabab` *as in Example 5.1.*

- $\mathscr{H}^{3}_{10} = 3$*; for* $u = w[10..12] =$ `baa`*, the factor* $w[3..9] =$ `bbabaab` *is the longest suffix of* $w[1..9]$ *that can be decomposed from right to left into the shortest prefixes of* $u$ — `bbabaab` = `b`$\cdot$`ba`$\cdot$`baa`$\cdot$`b`.
- $\mathscr{H}^{2}_{15} = 15$*; no prefix of* $u = w[15..16] =$ `bb` *matches a non-empty suffix of* $w[1..14]$.

The hook $\mathscr{H}_j^\ell$ has its utility when $j$ is a reference and $\ell = \mathsf{LUF}[j]$ as given in the following lemma.

**Lemma 5.3.** *If $\mathscr{H}_j^{\mathsf{LUF}[j]} = q$ then the following holds for all $i$, $1 \le i < j$, such that $\mathsf{LSF}_r[i] = j$ and $\mathsf{LSF}_\ell[i] \ge \mathsf{LUF}[j]$:*

$$\mathsf{LUF}[i] = \begin{cases} q - i & \text{if } i < q; \\ \mathsf{LUF}[j] & \text{otherwise.} \end{cases}$$

**Proof.** Let $u = w[j..j + \mathsf{LUF}[j] - 1]$ and $v = w[i..i + \mathsf{LSF}_\ell[i] - 1]$. Observe that $u$ occurs at position $i$ and that $v$ and $w[q..n]$ can be decomposed into unbordered prefixes of $u$.

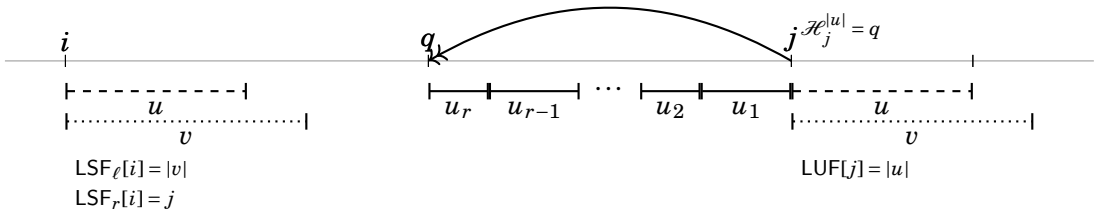


Figure 5.2: The unbordered-decomposition of $w[i..n]$ consists of $w[i..q-1]$ as the longest unbordered prefix; followed by a sequence of unbordered prefixes of $u$, including $u$ itself at position $j$. Therefore, $\mathsf{LUF}[i] = q - i$.

**Case a: i < q.** We shall prove that $w[i..q-1]$ is the longest unbordered prefix of $w[i..n]$; see Figure 5.2.

First, observe that any longer factor $w[i..k]$, $q \le k \le n$ has a suffix $w[q..k]$ which is composed of unbordered prefixes of $u$ (by Observation 5.2). This means that $w[i..k]$ must be bordered (because $u$ is its prefix).

To conclude, for a proof by contradiction suppose that $w[i..q-1]$ has a border $v'$. Note that $|v'| \le \mathsf{LSF}_\ell[i]$, so $v'$ is a prefix of $v$. Hence, it occurs both as a suffix of $w[1..q-1]$ and a prefix of $w[j..n]$, which contradicts the greedy construction of $q = \mathscr{H}_j^{|u|}$ (Observation 5.1) and thus definition of $q = \mathscr{H}_j^{|u|}$.

**Case b: i ≥ q.** The decomposition of $w[q..n]$ into unbordered prefixes of $u$ yields a decomposition of $w[i..n]$ into unbordered prefixes of $u$, starting with $u$. This is the unbordered-decomposition of $w[i..n]$ (see Proposition 5.2), which yields $\mathsf{LUF}[i] = |u| = \mathsf{LUF}[j]$. ■

## 5.4 Algorithm

The algorithm operates in two phases: a pre-processing phase followed by the main computation phase.

The following is accomplished in the pre-processing phase: Firstly, compute the longest successor factor array $\mathsf{LSF}_\ell$ together with $\mathsf{LSF}_r$ array. If $\mathsf{LSF}_r[i] = j$ then we say $i$ *refers to* $j$ and mark $j$ in a boolean array (IsReference) as a reference. Finally, initialise the array HOOK, that keeps the hook of each position, such that $\mathsf{HOOK}[i] = i$ (as $\mathscr{H}_i^0 = i$).

In the main computation phase, the algorithm computes the length of the longest unbordered factors for all positions in $w$. Moreover, it determines $\mathsf{HOOK}[j] = \mathscr{H}_j^{\mathsf{LUF}[i]}$ for each *potential reference*, i.e., each position $j$ such that $j = \mathsf{LSF}_r[i]$ and $\mathsf{LSF}_\ell[i] \geq \mathsf{LUF}[j]$ for some $i < j$; see Lemma 5.3.

Positions are processed from right to left (in decreasing order) so that if $i$ refers to $j$ then $\mathsf{LUF}[j]$ (and $\mathsf{HOOK}[j]$, if necessary) has already been computed before $i$ is considered. For each position $i$, the value of $\mathsf{LUF}[i]$ is updated as follows:

1. If $\mathsf{LSF}_\ell[i] = 0$ then $\mathsf{LUF}[i] = n - i + 1$. Observe that $i$ here is the first (from the right end) occurrence of the letter $w[i]$. If $i$ is also a reference (i.e. there is at least one other occurrence of the letter $w[i]$), it is called a *start reference*.

2. Otherwise

   a) If $\mathsf{LSF}_\ell[i] < \mathsf{LUF}[j]$ then $\mathsf{LUF}[i] = j + \mathsf{LUF}[j] - i$.

   b) If $\mathsf{LSF}_\ell[i] \geq \mathsf{LUF}[j]$ and $i \geq \mathsf{HOOK}[j]$ then $\mathsf{LUF}[i] = \mathsf{LUF}[j]$.

   c) If $\mathsf{LSF}_{\ell|}[i] \geq \mathsf{LUF}[j]$ and $i < \mathsf{HOOK}[j]$ then $\mathsf{LUF}[i] = \mathsf{HOOK}[j] - i$.

Subsequently, if $i$ is a potential reference then the algorithm also computes $\mathscr{H}_i^{\mathsf{LUF}[i]}$ to update $\mathsf{HOOK}[i]$. It is evident that the computational phase of the algorithm fundamentally reduces to finding the hooks for a subset of references, i.e. the set of potential references; for brevity, the term reference will mean a potential reference hereafter. Algorithm 5.1 presents the pseudo-code of this high level description of the algorithm; IsPotentialReference($i$) returns *true* if there exists $i'$ such that $\mathsf{LSF}_r[i'] = i$ and $\mathsf{LSF}_\ell[i'] \geq \mathsf{LUF}[i]$. The next subsection details the subroutine FindHook which constitutes the computational bulk of the algorithm.

**Algorithm 5.1** LONGESTUNBORDEREDFACTOR : Computes the Longest Unbordered Factor Array of the given word $w$ with length $n$.

```
1: function LONGESTUNBORDEREDFACTOR(w)
   ▷ Pre-processing:
2:     LSF_ℓ, LSF_r ← LONGESTSUCCESSORFACTOR(w)
3:     for i ← 1 to n do
4:         if LSF_ℓ[i] ≠ 0 then
5:             IsReference[LSF_r[i]] ← true
6:         end if
7:     end for
8:     HOOK[1..n] ← 1, ··· , n
   ▷ Main:
9:     for i ← n to 1 do
10:        if LSF_ℓ[i] = 0 then                ▷ possibly a start reference
11:            LUF[i] ← n − i + 1
12:        else
13:            j ← LSF_r[i]
14:            if LSF_ℓ[i] < LUF[j] then
15:                LUF[i] ← j + LUF[j] − i
16:            else if i ≥ HOOK[j] then
17:                LUF[i] ← LUF[j]
18:            else
19:                LUF[i] ← HOOK[j] − i
20:            end if
21:            if ISPOTENTIALREFERENCE(i) then
22:                HOOK[i] ← FINDHOOK(i)
23:            end if
24:        end if
25:    end for
26: end function
```

### 5.4.1 Finding Hook (Subroutine FINDHOOK)

**Main Idea** When FINDHOOK is called on a reference $j$, it returns $\mathscr{H}_j^{\mathsf{LUF[j]}}$. A simple greedy approach follows directly from Observation 5.1. Let $u = w[j..j + \mathsf{LUF}[j] - 1]$ and $\mathscr{H}_j^{\mathsf{LUF[j]}} = q$; inspect Figure 5.3. Initially, the factor $w[1..j-1]$ is considered and the shortest suffix of $w[1..j-1]$ which is a prefix of $u$ is computed. Let such prefix be $u_{i_1}$; observe that $u_{i_1}$ is unbordered. Then this suffix, $w[i_1..j-1] = u_{i_1}$ is truncated or chopped (conceptually) from the considered factor $w[1..j-1]$; the next factor considered will be $w[1..j-|u_{i_1}|-1]$. Thus, the subroutine iteratively computes

and truncates the shortest prefixes of $u$ from the right-end of the considered factor, shortening the length of the considered factor in each iteration and terminating as soon as no prefix of $u$ can be found. If the considered factor at termination is $w[1..q-1]$, position $q$ is returned by the subroutine as $\mathscr{H}_j^{\mathsf{LUF[j]}}$. We make a call to the subroutine FINDBETA$(q, j)$ (detailed in Subsection 5.4.3) to find the length of the shortest prefix of $w[j..j+\mathsf{LUF}[j]-1]$ ending at $q-1$.

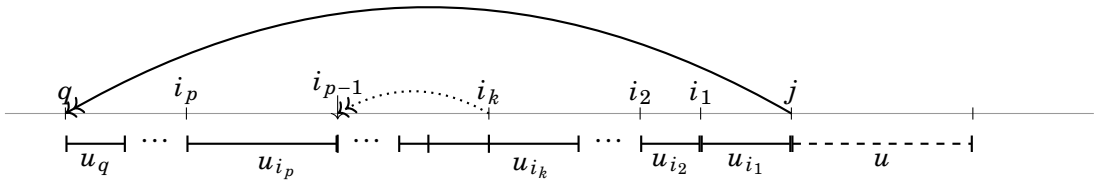


Figure 5.3: A chain of consecutive shortest prefixes of $u$ was computed at positions $i_1, i_2, \cdots i_k \cdots i_p$ and finally at position $q$. No prefix of $u$ is a suffix of $w[1..q-1]$. The hook value of position $j$ is then set to $q$, meanwhile, the hook of $i_k$ is set to $i_{p-1}$.

The factors considered by successive calls of FINDHOOK may overlap. Moreover, the same *chains* of consecutive shortest prefixes may be computed several times throughout the algorithm. To expedite the chain computation in the subsequent call to FINDHOOK on another reference $j'$ ($j' < j$), we can *recycle* some of the computations done for $j$ by shifting the value $\mathsf{HOOK}[\cdot]$ of each such index (at which a prefix was cut for $j$) leftwards (towards its final value). Consider the starting position $i_k$ at which $u_{i_k}$ was cut (i.e. $w[i_k..i_k+|u_{i_k}|-1] = u_{i_k}$ is the shortest unbordered prefix of $u$ computed for the factor $w[1..i_{k-1}-1]$). Let $i_p$ be the first position considered after $i_k$ such that $|u_{i_p}| > |u_{i_k}|$ (i.e. every prefix cut between $i_k$ and $i_p + |u_{i_p}|$ is an unbordered prefix of $u_{i_k}$). In other words, every factor $u_{i_k+1}, \ldots, u_{i_{p-1}}$ is a prefix of $u_{i_k}$; see Figure 5.3. Therefore, $w[i_{p-1}..i_k-1]$ can be decomposed into prefixes of $u_k$ i.e. the position $i_{p-1}$ represents $\mathscr{H}_{i_k}^{|u_{i_k}|}$. Consequently, we set $\mathsf{HOOK}[i_k]$ to $i_{p-1}$ so that the next time a prefix of length greater than or equal to $|u_k|$ is cut at $i_k$, we do not have to repeat truncating the prefixes $u_{k+1}, \ldots, u_{p-1}$ and we may start directly from position $i_{p-1}$.

**Implementation** Updating the hook values for these indices can be efficiently realised using a stack. Every starting position $i_p$, at which $u_{i_p}$ is cut, is pushed onto the stack as a *(length, position)* pair $(|u_{i_p}|, i_p)$. Before pushing, every element

$(|u_{i_k}|, i_k)$ such that $|u_{i_p}| > |u_{i_k}|$ is popped and the hook value of index $i_k$ is updated ($\mathsf{HOOK}[i_k] = \mathscr{H}_{i_k}^{|u_k|} = i_{p-1} = i_p + |u_{i_p}|$).

A key observation to make here is that throughout the algorithm, each unbordered prefix $u_{i_k}$ at position $i_k$ is computed just once by FINDHOOK. Nevertheless, a longer[ii] unbordered prefix $u'_{i_k}$ which is the shortest prefix of some other unbordered factor $u' \neq u$ may be computed at $i_k$ again when FINDHOOK is called on reference $j'$ ($u'$ is the longest unbordered factor at $j'$, where $q < j' < j$). Example 5.4 illustrates the functioning of the algorithm and FINDHOOK.

**Example 5.4.** *In the running example i.e. Example 5.1 ($w =$* `aabbabaabbaababbabab`*), the references (in the processing order) are* $-20, 19, 18, 17, 16, 15, 14, 11, 10$, *and* $7$*; all of these are the potential references except* $-7$ *(*$1$ *refers to it but* $LSF_\ell[1] = 5$ *and* $LUF[7] = 14$*) and* $11$ *(*$7$ *refers to it but* $LSF_\ell[7] = 3$ *and* $LUF[11] = 10$*). Only potential references will call* FINDHOOK*; out of these,* $20, 17, 16$, *and* $15$ *will have empty stacks. The hook and corresponding decomposition (and stacks; the left end is the bottom of the stack) of the rest of the references have been shown below.*

| **Reference** $i$ | $\mathsf{LUF}[i]$ (*luf*) | $\mathsf{HOOK}[i]$ | **Decomposition/Stack** |
|---|---|---|---|
| 19 | 2 (`ab`) | 17 | $w[17..18] =$ `ab` |
| | | | $(2, 17)$ |
| 18 | 2 (`ba`) | 13 | $w[13..17] =$ `ba`$\cdot$`b`$\cdot$`ba` |
| | | | $(2,16),(1,15),(2,13)$ |
| 14 | 3 (`abb`) | 1 | $w[1..13] =$ `a`$\cdot$`abb`$\cdot$`ab`$\cdot$`a`$\cdot$`abb`$\cdot$`a`$\cdot$`ab` |
| | | | $(2,12),(1,11),(3,8),(1,7),(2,5),(3,2),(1,1)$ |
| 10 | 3 (`baa`) | 3 | $w[3..9] =$ `b`$\cdot$`ba`$\cdot$`baa`$\cdot$`b` |
| | | | $(1,9),(3,6),(2,4),(1,3)$ |

FINDHOOK *also updates the hook values of the positions in the stacks; the final HOOK array is as follows:*

| $i$ | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $w[i]$ | a | a | b | b | a | b | a | a | b | b | a | a | b | a | b | b | a | b | a | b |
| $\mathsf{HOOK}[i]$ | 1 | 1 | 3 | 3 | 5 | 3 | 7 | 1 | 9 | 3 | 11 | 11 | 13 | 1 | 15 | 13 | 17 | 13 | 17 | 20 |

[ii] It will be easy to deduce after Lemma 5.5 that the length of the prefix cut (the next time) at the same position will be at least twice the length of the current prefix cut at it.

The positions in the stack of a reference can be partitioned based on the length of the prefix cut at it for that reference (i.e. the length which was pushed onto the stack paired with that position). This notion of partitioning – realised using *twin sets* (defined below) – aids in establishing the relationship between the stacks of various references. Let $\mathbb{S}_j$ be the set of positions pushed onto the stack during a call to FINDHOOK on reference $j$.

**Definition 5.2** (Twin Set)**.** A *twin set* of reference $j$ for length $\ell$, denoted by $\mathbb{T}_j^\ell$, is the set of all positions $i \in \mathbb{S}_j$ which were pushed onto the stack paired with length $\ell$ in the call to FindHook on reference $j$ i.e.

$$\mathbb{T}_j^\ell = \{i \mid (\ell, i) \text{ was pushed onto the stack of } j\}$$

**Example 5.5.** *The positions in the stack of the reference* 14 *as shown in Example 5.4 can be partitioned in three twin sets corresponding to the prefixes (of the luf at* 14 *which is* baa*) cut at those positions – prefixes with lengths* 1 *(*b*),* 2 *(*ba*), and* 3 *(*baa*) i.e.* $\mathbb{S}_{14} = \mathbb{T}_{14}^1 \bigcup \mathbb{T}_{14}^2 \bigcup \mathbb{T}_{14}^3$ *where*

$$\mathbb{T}_{14}^1 = \{11, 7, 1\},\ \mathbb{T}_{14}^2 = \{12, 5\} \text{ and } \mathbb{T}_{14}^3 = \{8, 2\}.$$

Note that a *unique* shortest unbordered prefix of $w[j\,..\,\mathsf{LUF}[j]-1]$ occurs at each $i$ belonging to the same twin set. However, as and when a longer prefix at $i$ is cut (say $\ell'$) for another reference $j' < j$, $i$ will be added to $\mathbb{T}_{j'}^{\ell'}$.

**Remark 5.2.** $\mathbb{S}_j = \bigcup_{\ell=1}^{LUF[j]} \mathbb{T}_j^\ell$.

Hereafter, a twin set will essentially imply a non-empty twin set. A pseudo-code implementation of FINDHOOK is given in Subroutine 5.2; the array InvTwinSet maintains a pointer to the most recent twin set that each position is in; the subroutines used by FINDHOOK have been spelled out in Table 5.1.

| **Subroutine** | **Input** | **Action** |
|---|---|---|
| NEWSTACK | - | Creates a new stack. |
| FINDBETA | Position $q$, Reference $j$ | Returns the length of the shortest prefix of $w[j\,..\,j+\mathsf{LUF}[j]-1]$ ending at $q-1$. |
| PUSH | Stack st, Pair $(\ell, i)$ | Pushes the given pair of length and position onto the stack. |

| | | |
|---|---|---|
| IsNotEmpty | Stack st | Returns **true** if the stack is not empty; **false** otherwise. |
| Top | Stack st | Returns the top element (here, a pair of length and position) of the stack. |
| Length | Pair $(\ell, i)$ | Returns the length (i.e. $\ell$) from the given pair of length and position. |
| Pop | Stack st | Pops the pair of length and position at the top of the stack and returns it. |

Table 5.1: Subroutines used by FindHook

**Subroutine 5.2** FindHook: Returns $\mathscr{H}_j^{\mathsf{LUF}[j]}$ and sets $\mathsf{HOOK}[i] \leftarrow \mathscr{H}_i^{\beta}$ for each $(\beta, i)$ pushed onto the stack of $j$.

```
1:  function FindHook(j)
2:      st ← NewStack( )
3:      q ← HOOK[j]
4:      β ← FindBeta(q, j)
5:      while (β ≠ 0) do
6:          HandlePopping(st, j, q, β)
7:          Push(st, (β, q − β))
8:          q ← HOOK[q − β]
9:          β ← FindBeta(q, j)
10:     end while
11:     HandlePopping(st, j, q, LUF[j] + 1)
12:     return q                                   ▷ returns H_j^{LUF[j]}
13: end function

14: function HandlePopping(st, j, q, β)
15:     while IsNotEmpty(st) and Length(Top(st)) < β do
16:         (length, pos) ← Pop(st)
17:         HOOK[pos] ← q                          ▷ q = H_pos^{length}
18:         InvTwinSet[pos] ← T_j^{length}
19:     end while
20: end function
```

### 5.4.2 Forest of Stacks

In this subsection, we establish the relationships amongst the stacks and twin sets of various references which are conducive to the analysis of the algorithm as well as to finding the length of the shortest prefix of the *luf* of a reference ending at a given position (computed by the subroutine FINDBETA). In doing so, we will make use of the following observation.

**Observation 5.3.** *$w[i..n]$, $\forall i \in \mathbb{S}_j$ admits a unique decomposition into unbordered prefixes of $w[j..j+\mathsf{LUF}[j]-1]$.*

**Lemma 5.4.** *If $j'$ is a reference such that $j' \in \mathbb{S}_j$, then $\mathscr{H}_{j'}^{\mathsf{LUF}[j']} \geq \mathscr{H}_{j}^{\mathsf{LUF}[j]}$.*

**Proof.** Let $u = w[j..j+\mathsf{LUF}[j]-1]$ and $u' = w[j'..j'+\mathsf{LUF}[j']-1]$. Since $j' \in \mathbb{S}_j$, the suffix $w[j'..n]$ can be decomposed into unbordered prefixes of $u$ (by Observation 5.3); in particular, any prefix of $u'$ can be decomposed into unbordered prefixes of $u$ (by Observation 5.2). Consequently, any decomposition into unbordered prefixes of $u'$ yields a decomposition into unbordered prefixes of $u$. In particular, $w[\mathscr{H}_{j'}^{\mathsf{LUF}[j']}..n]$ admits such a decomposition, which implies $\mathscr{H}_{j}^{\mathsf{LUF}[j]} \leq \mathscr{H}_{j'}^{\mathsf{LUF}[j']}$. ■

We observe that, throughout the algorithm, the stacks' creation follows a laminar structure. In the following, we present and prove three Lemmas (5.5, 5.6, and 5.7) which allow us to visualise the stacks of the references as a *forest*. If the stack $\mathbb{S}_j$ is the most recent stack containing a reference $j'$, we say that $j$ is the ***parent*** of $j'$. More formally, the parent of $j'$ is defined as $\min\{j \mid j' \in \mathbb{S}_j\}$. If a reference $j$ does not belong to any stack (i.e. has no parent), we will call it a ***base reference***. Consequently, each *tree* in the forest is related to a base reference $j$ such that the positions in $\mathbb{S}_j$ are partitioned into the corresponding twin sets $\mathbb{T}_j^{\ell}$ at the *root*. In general, the stack of a reference $j''$ whose parent is $j'$ (with its stack at level $l$) appears at level $l+1$ (appropriately partitioned into twin sets). See Example 5.6 for an illustration.

**Example 5.6.** *Consider a new word $w = (\texttt{aabaabbaabaabbb})^2$ (different from the running example). The associated arrays are as follows.*

| $i$ | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $w[i]$ | a | a | b | a | a | b | b | a | a | b | a | a | b | b | b | a | a | b | a | a | b | b | a | a | b | a | a | b | b | b |
| $\mathsf{LUF}[i]$ | 15 | 14 | 3 | 12 | 11 | 7 | 3 | 8 | 7 | 3 | 5 | 4 | 15 | 7 | 3 | 15 | 14 | 3 | 12 | 11 | 7 | 3 | 8 | 7 | 3 | 5 | 4 | 1 | 1 | 1 |
| $\mathsf{LSF}_{\ell}[i]$ | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 4 | 3 | 2 | 1 | 1 | 0 | 2 | 1 | 0 |
| $\mathsf{LSF}_{r}[i]$ | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 29 | 25 | 26 | 27 | 0 | 27 | 30 | 29 | 30 | 0 |

*Set of the references:* $\{30,29,28,27,26,25,24,23,22,21,20,19,18,17,16\}$

*Set of the potential references:* $\{30,29,28,27,26,25,24,23,22,21,20,19,18,17,16\}$

*Set of the base references:* $\{30,27,25\}$

*The trees of stacks corresponding to the base references have been shown in Figures 5.4 and 5.5.*

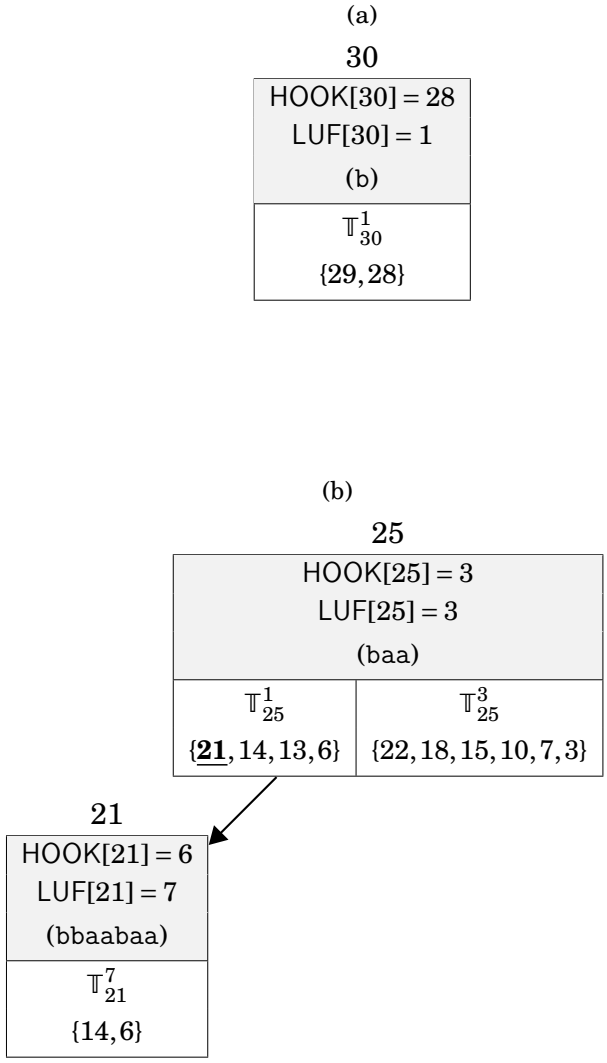


FIGURE 5.4. Figure illustrating the trees of stacks (partitioned into twin sets) corresponding to the base references 30 (Fig (a)) and 25 (Fig (b)). Only one reference pushes positions from its twin set onto its stack (shown in bold and is underlined). Note that the base reference 30 is also a start reference corresponding to the letter b.

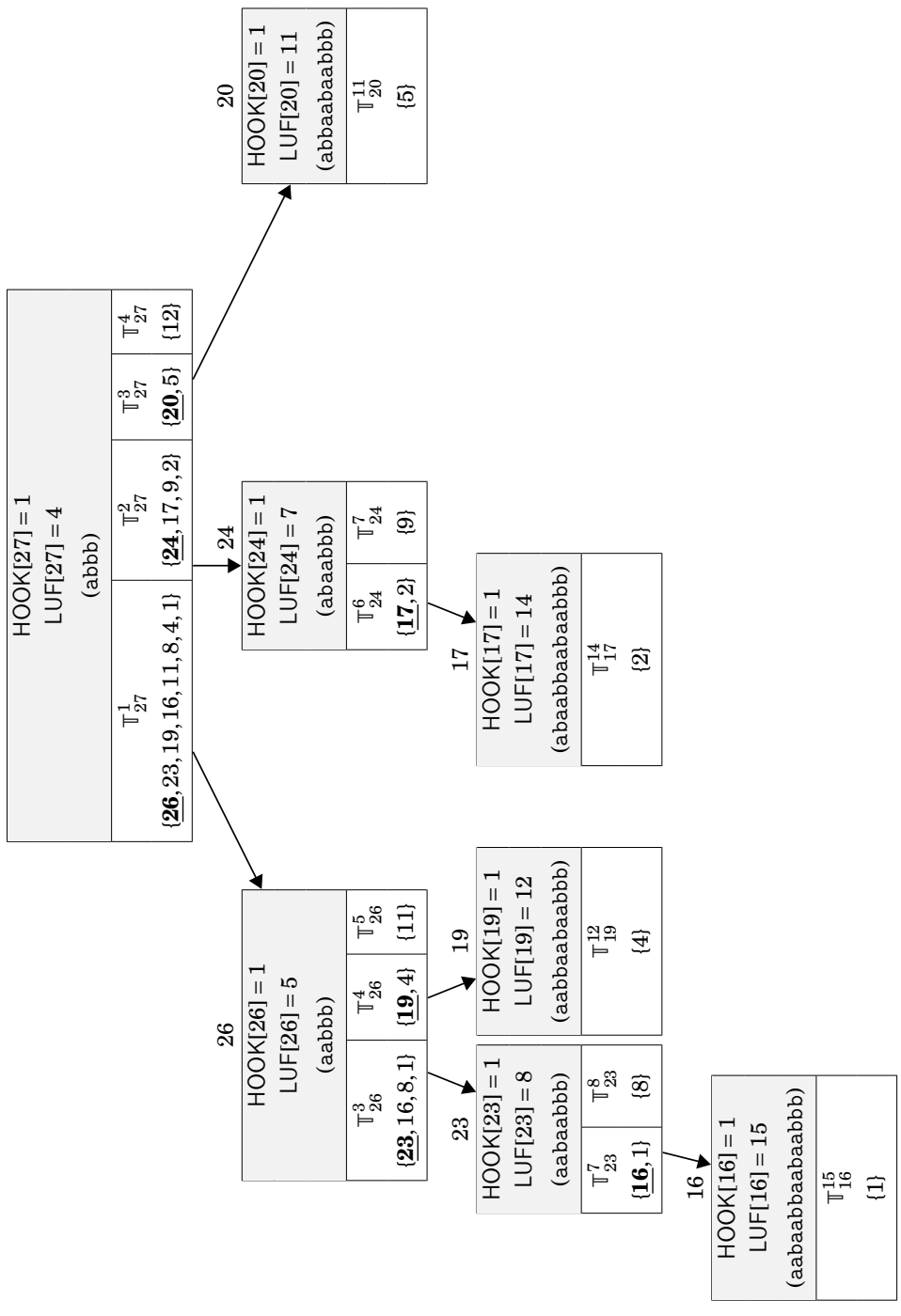


Figure 5.5: Figure illustrating the tree of stacks (partitioned into twin sets) corresponding to the base reference 27. References which end up pushing every position in their respective twin sets onto their respective stacks are shown in bold and are underlined. Note that the base reference 27 is also a start reference corresponding to the letter a.

**Lemma 5.5.** *If $j$ and $j'$ are two references such that $j$ is the parent of $j'$ and $j' \in \mathbb{T}_j^{\ell}$ for some $\ell < \mathsf{LUF}[j]$, then the following hold:*

1. $\mathbb{S}_{j'} \subset \mathbb{T}_j^{\ell}$ ;
2. *For each $i \in \mathbb{S}_{j'}$ there exists a $k$ which was added to $\mathbb{T}_j^{\ell'}$ , with $\ell' > \ell$, such that the pair $(k + \ell' - i, i)$ is pushed onto the stack of $j'$.*

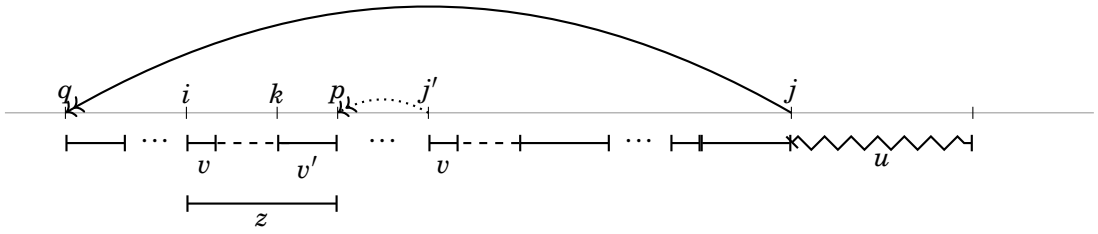


Figure 5.6: The pair $(|z|, i)$ is the first to be pushed onto the stack of $j'$. The factor $z$ is unbordered, has $v$ as a proper prefix and some $v'$ as a proper suffix, where both $v$ and $v'$ are unbordered prefixes of $u$ such that $|v| = \ell < |v'|$.

**Proof.** Let $u = w[j\,..\,j + \mathsf{LUF}[j] - 1]$, $u' = w[j'\,..\,j' + \mathsf{LUF}[j'] - 1]$, and $p$ be the value of $\mathsf{HOOK}[j']$ prior to the execution of FINDHOOK($j'$). Since $j' \in \mathbb{T}_j^{\ell}$, the earlier call FINDHOOK($j$) has set $\mathsf{HOOK}[j'] = \mathscr{H}_{j'}^{\ell}$. As $j$ is the parent of $j'$, no further call has updated $\mathsf{HOOK}[j']$. Thus, we conclude that $p = \mathscr{H}_{j'}^{\ell}$.

Consequently, the first pair pushed onto the stack of $j'$ (cf. Subroutine 5.2) is $(|z|, i)$, where $z = w[i\,..\,p-1]$ is the shortest suffix of $w[1\,..\,p-1]$ which also occurs as a prefix of $w[j'\,..\,n]$ (see Figure 5.6). Moreover, observe that $|z| > \ell$ by the greedy construction of $\mathscr{H}_{j'}^{\ell}$.

Recall that $j' \in \mathbb{T}_j^{\ell}$ implies that $w[j'\,..\,n]$ can be decomposed into unbordered prefixes of $u$ (by Observation 5.3), with the first prefix of length $\ell$, denoted $v = w[j'\,..\,j' + \ell - 1]$. With an occurrence at position $j'$, the factor $z$ also admits such a decomposition (by Observation 5.2), still with the first prefix $v$ (due to $|z| > |v|$). Additionally, note that $w[p\,..\,j' - 1]$ can be decomposed into unbordered prefixes of $v$. Concatenating the decompositions of $z = w[i\,..\,p-1]$, $w[p\,..\,j'-1]$, and $w[j'\,..\,n]$, we conclude that $w[i\,..\,n]$ can be decomposed into unbordered prefixes of $u$ with the first prefix (in this unique decomposition) equal to $v$. Hence, $i \in \mathbb{S}_{j'}$ belongs to the same twin set as $j'$; i.e., it satisfies the first claim of the lemma.

Additionally, in the aforementioned decomposition of $w[i..n]$ consider the factor $v' = w[k..p-1]$ which ends at position $p-1$. By the greedy construction of $\mathscr{H}^{\ell}_{j'}$, its length $|v'|$ is strictly larger than $\ell$, so $k \in \mathbb{T}^{\ell'}_{j}$ for $\ell' = |v'| > \ell$. Moreover, recall that $(|z|, i) = (k + \ell' - i, i)$ is pushed onto the stack of $j'$. Consequently, $i$ also satisfies the second claim of the lemma.

A similar reasoning is valid for each $i$ that will appear in $\mathbb{S}_{j'}$.

■

**Remark 5.3.** *If $j$ and $j'$ are two references such that $j' < j$ and $j' \in \mathbb{T}^{\mathsf{LUF}[j]}_{j}$ then $\mathbb{S}_{j'} = \emptyset$*

**Lemma 5.6.** *If $j$ is the parent of two references $j'' < j'$, both of which belong to $\mathbb{T}^{\ell}_{j}$, then*

$$\mathbb{S}_{j'} \cap \mathbb{S}_{j''} = \emptyset$$

**Proof.** The proof is trivial if $\ell = \mathsf{LUF}[j]$. Let $\ell < \mathsf{LUF}[j]$, $u = w[j..j + \mathsf{LUF}[j] - 1]$ and $v$ be the shortest unbordered prefix of $u$ cut at $j'$ and $j''$ (i.e., $|v| = \ell$). Let $u' = w[j'..j' + \mathsf{LUF}[j'] - 1]$ and $u'' = w[j''..j'' + \mathsf{LUF}[j''] - 1]$. Here, the current call to FINDHOOK function has been made on the reference $j''$. Consider the largest position $i$ such that it is common to the stacks of $j'$ and $j''$ i.e. $i \in \mathbb{S}_{j'}$ and $i \in \mathbb{S}_{j''}$. Let the prefixes cut at $i$ be $z_1 = w[i..p]$ and $z_2 = w[i..k]$. Observe that $i$ being the largest position and $j' \neq j''$ ensure that $|z_1| \neq |z_2|$. Without loss of generality, let $|z_1| < |z_2|$ (examine Figure 5.7).

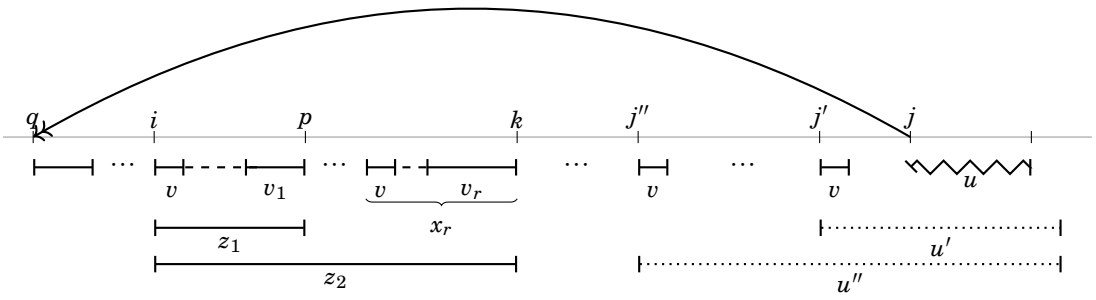


Figure 5.7: The pair $(|z_1|, i)$ and $(|z_2|, i)$ are pushed onto the stack of $j'$ and $j''$ where $i$ is a position common to both $\mathbb{S}_{j'}$ and $\mathbb{S}_{j''}$.

1. $j'$ **cuts** $z_2$ **and** $j''$ **cuts** $z_1$**:** We proceed with the proof by showing below that there is a reference between $j'$ and $j$ that pushes $j'$ onto its stack, thus contradicting the fact that $j$ is the parent of $j'$.

   overFollowing Observations 5.3 and 5.2, $w[i..k]$ can be decomposed into unbordered prefixes of $u''$ with the first prefix being $z_1$, i.e. $z_2 = z_1 \cdot x_1 \cdot x_2 \cdot \ldots \cdot x_r$. Here, $|x_r| > |z_1|$ otherwise $z_2$ is bordered. Moreover, each $x_i$ larger than $v$ has corresponding position in $\mathbb{S}_{j''}$ and others (i.e. $|x_i| \leq |v|$) are skipped because of HOOK[·]. Let $x_s$ be the first of these $x_i, 1 \leq i \leq r$ such that $|x_s| > |z_1|$. In the occurrence of $z_2$ at $j'$, let $j_0$ be the position corresponding to $x_s$ i.e. $j_0 = j' + |z_1 \cdots x_{s-1}|$. Note that $j_0 < j$ as $x_s$, like $z_1$ and each $x_i$ such that $|x_i| > |v|$, has $v$ as a proper prefix and some $v_i$ as a proper suffix where $v_i$ is an unbordered prefix of $u$ longer than $v$ (from Lemma 5.5).

   Now, we prove that $j_0$ is a (potential) reference. The fact that $j'$ is a potential reference ensures that $\tilde{u} = w[j_0..j' + |u'| - 1]$ is a repeated factor. Moreover, $\tilde{u}$ contains the *luf* at $j_0$, say $u_0$, because $u_0$ is a factor (or suffix) of $u'$ (since $w[j'..j_0 - 1]$ can be decomposed into prefixes of $x_s$); an implication is that $|\tilde{u}| \geq |u_0|$. Thus, $j_0$ is a reference if the last occurrence of $\tilde{u}$ is at $j_0$. For contradiction, assume that the factor $\tilde{u}$ has another occurrence at some position larger than $j_0$. This implies that there is another occurrence of $u$ after $j$ as $u_0$ contains $u$ (the *luf* at any position which is in the stack of $j$, ends at or after $j + |u| - 1$). This is not possible as the last of the occurrences of $u$ after $j$ would cause $j$, $j'$, $j''$ etc. to go onto its stack and $j$ would no longer be the parent of $j'$ or $j''$.

   Summing up, $j_0 < j$ is a reference with $x_s$ as a prefix of $u_0$. If $j$ is the parent of $j_0$ then $j_0$ would have pushed $j'$ onto its stack, otherwise another reference $j_{-1}$, $j_0 < j_{-1} < j$ that pushed $j_0$ onto its stack would have pushed $j'$ as well. In either case, $j$ is not the parent of $j'$ which is a contradiction.

2. $j'$ **cuts** $z_1$ **and** $j''$ **cuts** $z_2$**:** Using the similar argument as in Case 1, we can prove that this case leads to the conclusion that there is another reference between $j''$ and $j$ that would push $j''$ onto its stack and hence contradict that $j$ is the parent of $j''$.

■

**Lemma 5.7.** *If $j_1$ and $j_2$ are base references such that $j_1 \neq j_2$, then*

$$\mathbb{S}_{j_1} \cap \mathbb{S}_{j_2} = \emptyset$$

**Proof.** Suppose that the subroutine FINDHOOK is called for each position in $w$. We define a *base position* analogously as a position that does not appear in any stack. For a proof by contradiction, let $i$ be the largest element of $\mathbb{S}_{j_1} \cap \mathbb{S}_{j_2}$, with $(\ell_1, i)$ and $(\ell_2, i)$ pushed onto the stacks of $j_1$ and $j_2$, respectively. Let $j_1 > j_2$. Note that $i + \ell_1 \in \{j_1\} \cup \mathbb{S}_{j_1}$ and $i + \ell_2 \in \{j_2\} \cup \mathbb{S}_{j_2}$. Thus, our choice of $j_1 \neq j_2$ as base positions and $i$ as the largest element of $\mathbb{S}_{j_1} \cap \mathbb{S}_{j_2}$ guarantees $\ell_1 \neq \ell_2$.

Let $u$ be the longest unbordered factor at $j_1$. We first assume that $\ell_1 < \ell_2$. Note that due to the assumption that $i \in \mathbb{S}_{j_1}$, the suffix $w[i..n]$ can be decomposed into unbordered prefixes of $u$ (by Observation 5.3). In particular, $w[i..i+\ell_2-1]$ admits such a decomposition $w[i..i+\ell_2-1] = v_1 \cdots v_r$ with $|v_1| = \ell_1$. Moreover, observe that $|v_r| > \ell_1$, otherwise $v_r$ would be a border of $w[i..i+\ell_2-1]$.

Let $v_s$ be the first of these factors satisfying $|v_s| > \ell_1$ and let $k = j_2 + |v_1 \cdots v_{s-1}|$. Note that $w[j_2..k-1]$ admits a decomposition $w[j_2..k-1] = v_1 \cdots v_{s-1}$ into unbordered prefixes of $v_s$. Consequently, $j_2 \in \mathbb{S}_k$ if $k$ is a base position, and $j_2 \in \mathbb{S}_{k'}$ if $k$ is not a base position and $k \in \mathbb{S}_{k'}$ for some base position $k'$. In either case, this contradicts the assumption that $j_2$ is a base position. A similar line of argument contradicts the assumption that $j_1$ is a base position for the case when $\ell_1 > \ell_2$. Thus, base positions have disjoint stacks. Additionally, observe that the longest unbordered factor at some base position $j$, denoted as $u$, has the last occurrence at $j$ i.e. $u$ has no occurrence after $j$ (otherwise $j$ can not be a base position because it will be in the stack of the position of the last occurrence of $u$).

In fact, Algorithm 5.1 calls the subroutine FINDHOOK on a subset of positions, i.e. only on potential references. However, as we show below, all base references are actually base positions, hence their stacks are disjoint.

For a proof by contradiction, suppose that $j'$ is a base reference that would have been pushed onto the stack of a base position $j > j'$ if Algorithm 5.1 had not skipped $j$ (implying that $j$ is not a potential reference). This assumption entails that there is no occurrence of $u$ (where $u$ is the longest unbordered factor at $j$) at any position $k < j$ since $u$ has the last occurrence at $j$ and any previous occurrence would make $j$ a potential reference. Consequently, the longest unbordered factor at $j'$ has $u' = w[j'..j+|u|-1]$ as its prefix (as $u$ is unbordered and $u'$ can be decomposed into unbordered proper prefixes of $u$ followed by $u$). For $j'$ to be a potential reference, $u'$

must have an occurrence on its left. However, this means that $u$ has an occurrence at some $k < j' < j$, contrary to our assumption. ■

### 5.4.3 Finding the Shortest Border (Subroutine FINDBETA)

Given a reference $j$ and a position $q$, the subroutine FINDBETA returns the length $\beta$ of the shortest prefix of $w[j..j+\mathsf{LUF}[j]-1]$ that is a suffix of $w[1..q-1]$, or $\beta = 0$ if there is no such prefix; note that the sought shortest prefix is necessarily unbordered.

To find this length, we use 'prefix-suffix queries' of [KRRW15, KRRW12]. Such a query, given a positive integer $d$ and two factors $x$ and $y$ of $w$, reports all prefixes of $x$ of length between $d$ and $2d$ that occur as suffixes of $y$. The lengths of the sought prefixes are represented as an arithmetic progression, which makes it trivial to extract the smallest one. A single prefix-suffix query can be implemented in $\mathcal{O}(1)$ time and $\mathcal{O}(n)$ space after randomized pre-processing of $w$ which takes $\mathcal{O}(n)$ time in expectation [KRRW15], or $\mathcal{O}(n \log n)$ time with high probability [KRRW12]. Additionally, replacing the hash tables with the deterministic dictionaries [Ruž08], yields an $\mathcal{O}(n \log n \log^2 \log n)$-time deterministic pre-processing.

To implement FINDBETA, we set $x = [j..j+\mathsf{LUF}[j]-1]$, $y = [1..q-1]$ and we ask prefix-suffix queries for subsequent values $d = 1, 3, \ldots, 2^k - 1, \ldots$ until $d$ exceeds $\min(|x|, |y|)$. Note that we can terminate the search as soon as a query reports a non-empty answer. Hence, the running time is $\mathcal{O}(1 + \log \beta)$ if the query is successful (i.e., $\beta \neq 0$) and $\mathcal{O}(\log n)$ (as $\mathsf{LUF}[j] < n$) otherwise.

Furthermore, we can expedite the calls to FINDBETA if we already know that $\beta \notin \{1, \ldots, \ell\}$. In this case, the running time improves to $\mathcal{O}(1 + \log \frac{\beta}{\ell})$ because we can start the search with $d = \ell + 1$. Specifically, if $j$ is not a base reference and belongs to $\mathbb{T}^{\ell}_{j'}$ for some $j'$, we can start from $d = 2\ell + 1$ because Lemma 5.5.2 guarantees that $\beta \geq \ell + \ell' > 2\ell$.

## 5.5 Analysis

Algorithm 5.1 computes the longest unbordered factor at each position $i$; position $i$ is a start reference or it refers to some other position. The correctness of the computed $\mathsf{LUF}[i]$ follows directly from Lemmas 5.1 through 5.3.

### 5.5.1 Time Complexity

The analysis of the algorithm's running time necessitates probing of the total time consumed by FINDHOOK and the time spent by FINDBETA which, in turn, can be measured in terms of the *total size of the stacks* of various references (i.e. total number of push operations throughout the algorithm) .

**Lemma 5.8.** *The total size of all the stacks used throughout the algorithm is* $\mathcal{O}(n \log n)$[iii]. *Moreover, the total running time of the subroutine* FINDBETA *is* $\mathcal{O}(n \log n)$.

**Proof.** First, we shall prove that any position $p$ belongs to $\mathcal{O}(\log n)$ stacks.

By Lemma 5.5.1, the stack of any reference is a subset of the stack of its parent. Moreover, by Lemma 5.6, the stacks of references sharing the same parent are disjoint. A similar argument (presented in Lemma 5.7) shows that the stacks of the base references are disjoint.

Consequently, the references $j_1 > \ldots > j_s$ whose stacks $\mathbb{S}_{j_i}$ contain $p$ form a chain with respect to the parent relation: $j_1$ is a base reference, and the parent of any subsequent $j_i$ is $j_{i-1}$. Let us define $\ell_1, \ldots, \ell_s$ so that $p \in \mathbb{T}_{j_i}^{\ell_i}$. By Lemma 5.5.2, for each $1 \leq i < s$, there exist $k_i$ and $\ell_i' > \ell_i$ such that $k_i \in \mathbb{T}_{j_i}^{\ell_i'}$ and $\ell_{i+1} = k_i - p + \ell_i' \geq \ell_i + \ell_i' > 2\ell_i$. Due to $1 \leq \ell_i \leq n$, this yields $s \leq 1 + \log n = \mathcal{O}(\log n)$, as claimed.

Next, let us analyse the successful calls to FINDBETA such that FINDBETA$(q, j)$ returns $\beta$ where $\beta > 0$ and $p = q - \beta$. Observe that after each such call, $p$ is inserted to $\mathbb{S}_j$ and to the twin set $\mathbb{T}_j^{\beta}$ , i.e, $j = j_i$ and $\beta = \ell_i$ for some $1 \leq i \leq s$. Moreover, if $i > 1$, then $j_i \in \mathbb{T}_{j_{i-1}}^{\ell_{i-1}}$ , which we are aware of while calling FINDBETA. Hence, we can make use of the fact that $\ell_i \notin \{1, \ldots, 2\ell_{i-1}\}$ to find $\beta = \ell_i$ in time $\mathcal{O}(\log \frac{\ell_i}{\ell_{i-1}})$. For $i = 1$, the running time is $\mathcal{O}(1 + \log \ell_1)$. Hence, the overall running time of successful queries FINDBETA$(q, j) = \beta$ with $p = q - \beta$ is $\mathcal{O}(1 + \log \ell_1 + \sum_{i=2}^{s} \log \frac{\ell_i}{\ell_{i-1}}) = \mathcal{O}(1 + \log \ell_s) = \mathcal{O}(\log n)$, which sums up to $\mathcal{O}(n \log n)$ across all positions $p$.

As far as the unsuccessful calls (FINDBETA$(q, j) = 0$) are concerned, we observe that each such call terminates the enclosing execution of FINDHOOK. Hence, the number of such calls is bounded by $n$ and their overall running time is clearly $\mathcal{O}(n \log n)$. ■

[iii] See Appendix A for an alternative intuitive proof.

**Theorem 5.1.** *Given a word $w$ of length $n$, Algorithm 5.1 solves the* LONGEST UNBORDERED FACTOR ARRAY *problem in $\mathcal{O}(n \log n)$ time with high probability. It can also be implemented deterministically in $\mathcal{O}(n \log n \log^2 \log n)$ time.*

**Proof.** Assuming an integer alphabet, the computation of $\mathsf{LSF}_\ell$ and $\mathsf{LSF}_r$ arrays along with the constant time per position initialisation of the other arrays sum up the pre-processing stage (Lines 2–8) to $\mathcal{O}(n)$ time. The running time required for the assignment of the luf for all positions (Lines 9–20) is $\mathcal{O}(n)$. The time spent in construction of the data structure to answer prefix-suffix queries used in FINDBETA is $\mathcal{O}(n \log n)$ with high probability or $\mathcal{O}(n \log n \log^2 \log n)$ deterministic.

Additionally, the total running time of the subroutine FINDHOOK for all the references, being proportional to the aggregate size of all the stacks, can be deduced from Lemma 5.8. This has been shown to be $\mathcal{O}(n \log n)$ in the worst case, same as the total running time of FINDBETA. The claimed bound on the overall running time follows. ■

### 5.5.2 Space Complexity

Analysis of the space used by the algorithm is straightforward – all the arrays $(\mathsf{LSF}_r, \mathsf{LSF}_\ell, \mathsf{LUF}, \mathsf{HOOK})$ and the data-structure to answer the FINDBETA queries consume linear space with respect to the length of $w$. The total space taken up by all the stacks (and twin sets) are upper bounded by $\mathcal{O}(n \log n)$ (from Lemma 5.8). Overall, $\mathcal{O}(n \log n)$ space is required by the algorithm.

### 5.5.3 Words Exhibiting Worst-Case Behaviour

To show that the upper bound shown in Lemma 5.8 (consequently Theorem 5.1) in the worst case is tight, we design an infinite family of words that exhibit the worst-case behaviour.

A word can be made to exhibit the worst-case behaviour if we force the maximum number of positions to be pushed onto $\Theta(\log n)$ stacks. This can be achieved as follows.

1. Maximize the number of references: Every position in each twin set $\mathbb{T}_j^l$ is a reference.

2. Maximize the size of each stack: The largest position (reference) in any twin set pushes the rest of the positions onto its stack. If $j'$ is largest reference in $\mathbb{T}_j^\ell$ then $\mathbb{S}_{j'} = \mathbb{T}_j^\ell - \{j'\}$.

3. Maximize the number of twin sets obtained from a stack: This increases the number of unbordered prefixes that can be cut at some position $i$, therefore, increasing the number of repushes of $i$. This can be achieved by keeping $|\mathbb{T}_j^\ell| = 2|\mathbb{T}_j^{\ell+1}| + 1$.

Using the above, Algorithm 5.3 creates a word $w$ over $\Sigma = \{\mathtt{a}, \mathtt{b}\}$, such that the total size of the stack of the base reference $j$ ($w[j] = \mathtt{a}$) and the references that appear in $\mathbb{S}_j$ is $\mathcal{O}(\log n)$.

Consider, for instance, the following words (generated by Algorithm 5.3) exhibiting the maximum total size of the stacks used: $w_3 = (\mathtt{aabaabb})^2$, $d = 3$; $w_4 = (\mathtt{aabaabbaabaabbb})^2$, $d = 4$; $w_5 = (\mathtt{aabaabbaabaabbbaabaabbaabaabbbb})^2$, $d = 5$; etc., where $d$ is the maximum number of stacks onto which some proportional number of elements have been pushed by Algorithm 5.1. Position 1 in $w_4$, for example, is pushed onto four stacks paired with lengths 1, 3, 7, and finally 15; the deepest tree of stacks corresponding to $w_4$ has been shown in Figure 5.5. The total size of the stacks used by each word from this family of words is thus $\Theta(n \log n)$. Figure 5.8 shows the logarithmic increase (coloured blue) of the maximum number of stacks ($d$) onto which some element gets pushed as $n$ increases for the specified family of the words. Moreover, the almost linear appearing red plot in the same figure exhibits how the total size of the stacks used by Algorithm 5.1 grows with the length of these specially designed words.

**Algorithm 5.3** Create Word $w_d$ over $\Sigma = \{\mathtt{a}, \mathtt{b}\}$ for a given $d$.

```
1: w ← ε
2: block ← "a"
3: for i ← 1 to d − 1 do
4:     w ← w + block + w
5:     block ← block + "b"
6: end for
7: w ← w + block
8: w ← w + w
```

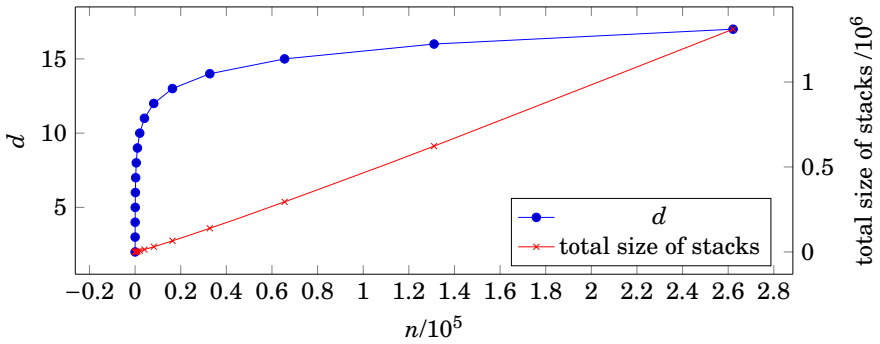


Figure 5.8: Plot showing the maximum number of stacks onto which some element has been pushed ($d$) and the total size of the stacks used by Algorithm 5.1 for specially designed words.

## 5.6 Practical Enhancement

In this section, we present an observation that provides a technical short-cut to speed up the computations in practice, although it does not affect the asymptotic time-bounds of the algorithm. This short-cut avoids the computations for finding hooks for the references which are not the centre of a *square*. A square is a word of the form $uu$; the first position of the second $u$ is called *the centre* of the square.

### Square Array

The *square array* (SQ) specifies, for each position $i$ in $w$, whether there is a square centred at $i$. Formally, the array SQ is defined as follows.

$$\mathsf{SQ}[i] = \begin{cases} 1 & \text{if } \exists\, \ell > 0 \text{ such that } w[i\,..\,i+\ell-1] = w[i-\ell\,..\,i-1], \\ 0 & \text{otherwise.} \end{cases}$$

For integer alphabets, we can compute for each position in $w$, the length of the shortest square centered at this position (also known as the *local period*) in $\mathcal{O}(n)$ time [DKK+04]. Thus, we can compute the array SQ in the pre-processing phase trivially from this array of local periods.

**Example 5.7.** *Consider the same word* $w =$ `aabbabaabbaababbabab` *as in Example 5.2. The associated square array is as follows.*

| $i$ | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $w[i]$ | a | a | b | b | a | b | a | a | b | b | a | a | b | a | b | b | a | b | a | b |
| SQ[$i$] | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 0 |

**Lemma 5.9.** *If a position $j$ is not the centre of any square (i.e.* SQ[$j$] = 0*) then $\mathscr{H}_j^\ell = j$, for all $\ell$ such that* $0 \le \ell \le$ LUF[$j$].

**Proof.** Clearly, for $\ell = 0$, $\mathscr{H}_j^\ell = j$. For $0 < \ell \le$ LUF[$i$], assume that $\mathscr{H}_j^\ell = q$ such that $1 \le q < j$. If $u = w[j..j+\ell-1]$ then let $u_1$ be the last prefix in the decomposition of $w[q..j-1]$ into unbordered prefixes of $u$. In this case, $j$ is the centre of the shortest square $u_1u_1$ (i.e. SQ[$j$] $\neq 0$) which is a contradiction. Therefore, $q = j$. ■

As a consequence of Lemma 5.9, if $i$ is a potential reference such that SQ[$i$] = 0 then HOOK[$i$] = $i$. For instance, in the running example (see Example 5.4) each of the references 20, 17, and 15 is a potential reference but is not the centre of some square and thus the algorithm could avoid calling FINDHOOK on these references. In other words, the algorithm can speed up by computing the $\mathscr{H}_i^{\text{LUF}[i]}$ to update HOOK[$i$] by calling FINDHOOK only when $i$ is a potential reference such that SQ[$i$] $\neq 0$.

CHAPTER 6

# CONCLUDING REMARKS

This dissertation presented three algorithms for problems – pertaining to macro-level uncertainty and local regularity in strings – that arise in the context of genomic sequence analysis. Below, we provide a brief summary of the work covered in this dissertation and discuss some open problems and possible research directions within the same framework as that of the presented work.

In Chapter 3, we presented an optimal i.e. $\mathcal{O}(n+m)$ time and space algorithm to compute all superbubbles in a directed acyclic graph, where $n$ is the number of vertices and $m$ is the number of edges, improving the best-known $\mathcal{O}(m \log m)$-time algorithm for this problem [SSS+15]. It is also interesting to note that in this type of graphs, that is, those constructed from sequences over a fixed-sized alphabet, the out-degree of each vertex is bounded by the size of the alphabet (four for DNA alphabet) and therefore the time complexity of the proposed algorithm is essentially linear in $n$. Structures generalising superbubbles like *ultrabubbles* and *snarls* (for bidirected and biedged graphs) have already been proposed [PNGH17]. Superbubbles and other generalisations provide a basis for specifying the sites and alleles in the graphical representation of a reference genomic cohort (showing genetic variations). However, not every site corresponds to these structures. One possible research direction could be to identify more general classes of subgraphs that could cover the parts of the graph not falling under superbubbles and its analogous structures.

Motivated by the necessity of alternative representations of a reference sequence for population-based genome assembly, in Chapter 4 we introduced and formalised

the notion of elastic-degenerate strings. In particular, we presented a practically-efficient algorithm for pattern matching in elastic-degenerate texts. Given a solid pattern and an elastic-degenerate text, the presented algorithm runs in $\mathcal{O}(N+\alpha\gamma mn)$ time, where $m$ is the length of the given pattern, $n$ and $N$ are the length and total size of the given elastic-degenerate text, respectively, and $\alpha$ and $\gamma$ are the parameters, respectively representing the maximum number of strings in any elastic-degenerate symbol of the text and the maximum number of elastic-degenerate symbols that any occurrence of the pattern may span in the text. Note that in applications like intra-species genetic variations studies, the pattern is a read, the text is the reference cohort of the population, $\alpha$ represents the number of sequences in the reference cohort, and $\gamma$ represents the number of genetic variation-sites falling in a full occurrence. The values of these parameters are usually small and so the presented algorithm is expected to work very fast in practice (as has been corroborated by the presented experiments using synthetic data-sets). The space used by the algorithm is linear in the size of the input. Elastic-degenerate strings as a model have opened up a new line of research and several improved algorithms (with changed definitions of an occurrence) have already been proposed [GIL$^+$17, BPPR17, ANI$^+$18, PR18, CGH18]. From a theoretical perspective, this model could potentially be adapted for classical string-problems (other than the pattern matching problem) like compression, finding regularities etc.

In Chapter 5, an algorithm to compute the *Longest Unbordered Factor Array* of a given word $w$ for general alphabets has been presented, with a time-complexity of $\mathcal{O}(n\log n)$ with a high probability (or $\mathcal{O}(n\log n\log^2\log n)$ deterministic), where $n$ is the length of $w$. This array specifies the length of the maximal unbordered factor starting at each position of $w$. This is a major improvement on the running time of the previously best worst-case algorithm working in $\mathcal{O}(n^{1.5})$ time for integer alphabets [Gawrychowski et al., 2015]. We also showed that the analysis of our algorithm is tight: an infinite family of words that exhibit the worst-case behaviour of the algorithm was described in this chapter. We would like to highlight that the Hook data-structure proposed in this chapter can be computed in a modular way i.e. without referring to the Longest Unbordered Factor Array; calling the subroutine to find the hook of each position $i$ i.e. $\mathscr{H}_i^{w[i..n]}$ can be achieved in the same time-bounds and thus can be used as an independent data-structure. Despite the theoretical origin of this problem, because of the close association of borders with regularities in strings, it may find applications in computational biology owing to the highly

repetitive nature of genomic sequences. One possible avenue for this research is to characterise a string using its Longest Unbordered Factor Array and the associated Hook data-structure that we proposed in this chapter (Chapter 5). For example, we observe that if $u$ is the longest unbordered factor at some position $i$ of a word $w$ and position $q$ is $\mathscr{H}_i^{|u|}$ then $w[q\,..\,n]$ can be decomposed into prefixes of $u$ and a careful selection of such positions may result in a compressed representation of a string. Nevertheless, from a purely theoretical viewpoint, computing the longest unbordered factor in $\mathcal{O}(n)$ time for integer alphabets remains an open problem.

Moreover, each of the presented algorithms has been implemented and the corresponding tool along with its source code have been made publicly available (https://github.com/Ritu-Kundu). It is worth mentioning that the rationale for implementation is different for each algorithm as described below:

- The algorithm presented in Chapter 3 for finding superbubbles has a direct application in identifying and defining sites in a reference graph. Its corresponding tool was developed in the hope that it may be of use to the bioinformatics community.

- The theoretical time bound of the algorithm for pattern matching in an elastic-degenerate string (proposed in Chapter 4) suggests that the algorithm will be impractical for large values of the parameters governing the running time. However, in practice the parameters are usually small. An implementation of the algorithm was required so that experiments could be conducted on data-sets having parameter values similar to those in real data, in order to corroborate that the algorithm is practical for real data-sets.

- The algorithm presented in Chapter 5 solves the problem of finding the longest unbordered factor array of a given word which is mainly of theoretical interest. An abstract analysis was proving to be insufficient owing to the large sizes of the *interesting* input instances and multiple inter-dependent factors controlling the behaviour of the algorithm. The implementation, therefore, was done to enable a better understanding of how a partially developed solution progresses and to gain detailed insights into its limitations and problems when the algorithm was still being developed.

APPENDIX A

# AN ALTERNATIVE PROOF OF LEMMA 5.8

Below, a more verbose and intuitive proof of Lemma 5.8 is given. For a reference $j'$ with its parent $j$ such that $j' \in \mathbb{T}_j^\ell$ , $|\mathbb{S}_{j'}|$ depends on three factors (following Lemma 5.5 and Lemma 5.6):

1. $|\mathbb{T}_j^\ell|$: every position in $\mathbb{S}_{j'}$ will come from $\mathbb{T}_j^\ell$ (Lemma 5.5 (1)).
2. The number of references in $\mathbb{T}_j^\ell$: $\mathbb{T}_j^\ell$ is partitioned into disjoint stacks; each corresponds to a distinct reference in $\mathbb{T}_j^\ell$ (Lemma 5.6).
3. $\sum_{i=\ell+1}^{\mathsf{LUF}[j]} |\mathbb{T}_j^i|$: the distribution of positions in the twin sets corresponding to lengths greater than $\ell$ also decides the positions in $\mathbb{S}_{j'}$ (Lemma 5.5 (2)).

The computations done by FINDHOOK when called on a base reference $j$ and all references $j' \in \mathbb{S}_j$, is directly proportional to the size of the corresponding tree. The following corollaries assist in determining the upper bound on the depth of the corresponding tree and hence its size; see Lemma A.1 below.

**Corollary A.1.** *If $j$ and $j'$ are the references such that $j$ is the parent of $j'$ then*

$$|\mathbb{S}_{j'}| < |\mathbb{S}_j|/2$$

**Proof.** Lemma 5.5 states that for each $i \in \mathbb{S}_{j'}$ there exists a $k$ which was added to $\mathbb{T}_j^{\ell'}$, with $\ell' > \ell$, such that the pair $(k + \ell' - i, i)$ is pushed onto the stack of $j'$. As a result, $i, k \in \mathbb{S}_j$ while $k \notin \mathbb{S}_{j'}$ yielding $|\mathbb{S}_{j'}| < |\mathbb{S}_j|/2$ (note that $j'$ itself is in $\mathbb{S}_j$ but not in $\mathbb{S}_{j'}$). ■

**Lemma A.1.** *For a base reference $j$,*

$$\sum_{j' \in \mathbb{R} \cap \mathbb{S}_j} |\mathbb{S}_{j'}| < |\mathbb{S}_j| \log |\mathbb{S}_j|$$

*where $\mathbb{R}$ is the set of all references.*

**Proof.** Consider a base reference $j$ such that $\mathbb{T}_j^{\ell_1}, \mathbb{T}_j^{\ell_2}, \cdots, \mathbb{T}_j^{\ell_t}$ are the (non-empty) twin sets obtained from $\mathbb{S}_j$ and $\ell_1 < \ell_2 < \cdots < \ell_t$. An upper bound on the size of the tree associated with $j$ can be obtained if we maximize the depth of the tree while populating the tree to the maximum at every level in the following way:

1. Every position $i \in \mathbb{T}_j^{\ell}$ is a reference.
2. The largest position (reference) in any twin set at any level pushes every other position onto its stack.

This constrains the total stack size at any level to be less than $|\mathbb{S}_j|$. From Lemma 5.6 the stacks of the references sharing the same parent are disjoint. Using the fact that stacks at the same level within a tree are disjoint along with Corollary A.1, it can be inferred that the depth is no more than $\log |\mathbb{S}_j|$.

Thus, the total size of the tree (consequently, the total sizes of the stacks of all the references in $\mathbb{S}_j$, excluding that of the base reference $j$) is less than $|\mathbb{S}_j| \log |\mathbb{S}_j|$. ■

**Lemma A.2.** *The total size of all the stacks used by Algorithm 1 is $\mathcal{O}(n \log n)$.*

**Proof.** Let $\mathbb{R}$ and $\mathbb{B} \subseteq \mathbb{R}$ be the set of all references and all base references, respectively. For two references $j_1, j_2 \in \mathbb{B}$ we have $\mathbb{S}_{j_1} \cap \mathbb{S}_{j_2} = \emptyset$ (From Lemma 5.7) i.e. stacks at the root of different trees are disjoint. Additionally, $\sum_{j \in \mathbb{B}} |\mathbb{S}_j| < n$. By summing the result from Lemma A.1 for the set of all base references, we complete the proof. ■

# BIBLIOGRAPHY


[AAH+15] A. Alatabbi, S. Azmin, M. K. Habib, C. S. Iliopoulos, and M. S. Rahman.
Simplisms: A simple, lightweight and fast approach for structured motifs searching.
In *Bioinformatics and Biomedical Engineering - Third International Conference*, pages 219–230, 2015.

[Abr87] K. Abrahamson.
Generalized string matching.
*SIAM Journal on Computing*, 16(6):1039–1051, 1987.

[AFG+94] A. Amir, M. Farach, Z. Galil, R. Giancarlo, and K. Park.
Dynamic dictionary matching.
*Journal of Computer and System Sciences*, 49(2):208 – 222, 1994.

[AFI91] A. Apostolico, M. Farach, and C. S. Iliopoulos.
Optimal superprimitivity testing for strings.
*Information Processing Letters*, 39(1):17 – 20, 1991.

[AHU87] A. V. Aho, J. E. Hopcroft, and J. D. Ullman.
The design and analysis of computer algorithms, 1974.
*Reading: Addison-Wesley*, pages 207–209, 1987.

[AKO02] M. I. Abouelhoda, S. Kurtz, and E. Ohlebusch.
The enhanced suffix array and its applications to genome analysis.
In R. Guigó and D. Gusfield, editors, *Algorithms in Bioinformatics: Second International Workshop, WABI 2002*, pages 449–463, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.

[AL62] G. Adel'son-Vel'skii and E. M. Landis.
An algorithm for organization of information.

*Dokl. Akad. Nauk SSSR*, 146(2):263–266, 1962.

[ANI+18] K. Aoyama, Y. Nakashima, T. I, S. Inenaga, H. Bannai, and M. Takeda.
Faster Online Elastic Degenerate String Matching.
In G. Navarro, D. Sankoff, and B. Zhu, editors, *Annual Symposium on Combinatorial Pattern Matching (CPM 2018)*, volume 105 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 9:1–9:10, Dagstuhl, Germany, 2018. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[Bal11] S. Balasubramanian.
Sequencing nucleic acids: from chemistry to medicine.
*Chem Commun (Camb)*, 47(26):7281–7286, Jul 2011.

[Bat05] S. Batzoglou.
Algorithmic challenges in mammalian genome sequence assembly.
*Encyclopedia of genomics, proteomics and bioinformatics, John Wiley and Sons, Hoboken (New Jersey)*, 2005.

[BCS+17] A. Bhardwaj, B. Cizmeci, E. Steinbach, Q. Liu, M. Eid, J. AraUjo, A. E. Saddik, R. Kundu, X. Liu, O. Holland, M. A. Luden, S. Oteafy, and V. Prasad.
A candidate hardware and software reference setup for kinesthetic codec standardization.
In *2017 IEEE International Symposium on Haptic, Audio and Visual Environments and Games (HAVE)*, pages 1–6, Oct 2017.

[BdD14] H. Buermans and J. den Dunnen.
Next generation sequencing technology: Advances and applications.
*Biochimica et Biophysica Acta (BBA) - Molecular Basis of Disease*, 1842(10):1932 – 1941, 2014.
From genome to function.

[BFC00] M. A. Bender and M. Farach-Colton.
The LCA problem revisited.
In *Proceedings of the Latin American Symposium on Theoretical Informatics LATIN*, pages 88–94, 2000.

[BIK+15] C. Barton, C. S. Iliopoulos, R. Kundu, S. P. Pissis, A. Retha, and F. Vayani.
Accurate and efficient methods to improve multiple circular sequence alignment.
In E. Bampis, editor, *Experimental Algorithms: 14th International Symposium, SEA 2015, Paris, France, June 29 – July 1, 2015, Proceedings*, pages 247–258, Cham, 2015. Springer International Publishing.

[BIK+16] L. Brankovic, C. S. Iliopoulos, R. Kundu, M. Mohamed, S. P. Pissis, and F. Vayani.
Linear-time superbubble identification algorithm for genome assembly.
*Theoretical Computer Science*, 609, Part 2:374 – 383, 2016.

[BKPR16] C. Barton, T. Kociumaka, S. P. Pissis, and J. Radoszewski.
Efficient Index for Weighted Sequences.
In R. Grossi and M. Lewenstein, editors, *27th Annual Symposium on Combinatorial Pattern Matching (CPM 2016)*, volume 54 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 4:1–4:13, Dagstuhl, Germany, 2016. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[BLGVW10] P. Bille, I. Li Gørtz, H. W. Vildhøj, and D. K. Wind.
String matching with variable length gaps.
In E. Chavez and S. Lonardi, editors, *String Processing and Information Retrieval*, pages 385–394, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[BM77] R. S. Boyer and J. S. Moore.
A fast string searching algorithm.
*Commun. ACM*, 20(10):762–772, October 1977.

[BMK+08] J. Butler, I. MacCallum, M. Kleber, I. A. Shlyakhter, M. K. Belmonte, E. S. Lander, C. Nusbaum, and D. B. Jaffe.
ALLPATHS: de novo assembly of whole-genome shotgun microreads.
*Genome Research*, 18(5):810–820, 2008.

[BP18] C. Barton and S. P. Pissis.
Crochemore's partitioning on weighted strings and applications.
*Algorithmica*, 80(2):496–514, 2018.

[BPPR17] G. Bernardini, N. Pisanti, S. P. Pissis, and G. Rosone.
Pattern matching on elastic-degenerate text with errors.
In G. Fici, M. Sciortino, and R. Venturini, editors, *String Processing and Information Retrieval: 24th International Symposium, SPIRE 2017, Palermo, Italy, September 26–29, 2017, Proceedings*, volume 10508 of *Lecture Notes in Computer Science*, pages 74–90, Cham, 2017. Springer International Publishing.

[BS12] F. BLANCHET-SADRI.
Algorithmic combinatorics on partial words.
*International Journal of Foundations of Computer Science*, 23(06):1189–1206, 2012.

[BSBDW17] F. Blanchet-Sadri, M. Bodnar, and B. De Winkle.
New bounds and extended relations between prefix arrays, border arrays, undirected graphs, and indeterminate strings.
*Theory of Computing Systems*, 60(3):473–497, Apr 2017.

[BYG92] R. Baeza-Yates and G. H. Gonnet.
A new approach to text searching.
*Commun. ACM*, 35(10):74–82, October 1992.

[CFOS04] A. M. Carvalho, A. T. Freitas, A. L. Oliveira, and M.-F. Sagot.
*String Processing and Information Retrieval: 11th International Conference, SPIRE 2004, Padova, Italy, October 5-8, 2004. Proceedings*, chapter Efficient Extraction of Structured Motifs Using Box-Links, pages 267–268.
Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.

[CGH18] A. Cisłak, S. Grabowski, and J. Holub.
Sopang: online text searching over a pan-genome.
*Bioinformatics*, page bty506, 2018.

[Cha05] E. Y. Chan.
Advances in sequencing technology.

*Mutation Research / Fundamental and Molecular Mechanisms of Mutagenesis*, 573(1):13 – 40, 2005.
Single Nucleotide Polymorphisms (SNPs): Detection, Interpretation, and Applications.

[CHL07] M. Crochemore, C. Hancart, and T. Lecroq.
*Algorithms on Strings*.
Cambridge University Press, 2007.
392 pages.

[CI08] M. Crochemore and L. Ilie.
Computing longest previous factor in linear time and applications.
*Inf. Process. Lett.*, 106(2):75–80, 2008.

[CII+13] M. Crochemore, L. Ilie, C. S. Iliopoulos, M. Kubica, W. Rytter, and T. Waleń.
Computing the longest previous factor.
*Eur. J. Comb.*, 34(1):15–26, 2013.

[CIK+10] M. Crochemore, C. S. Iliopoulos, M. Kubica, W. Rytter, and T. Waleń.
Efficient algorithms for two extensions of LPF table: The power of suffix arrays.
In *Proceedings of the Conference on Current Trends in Theory and Practice of Comp. Sci. SOFSEM*, pages 296–307, 2010.

[CIK+16a] M. Crochemore, C. S. Iliopoulos, T. Kociumaka, R. Kundu, S. P. Pissis, J. Radoszewski, W. Rytter, and T. Waleń.
Near-optimal computation of runs over general alphabet via non-crossing lce queries.
In S. Inenaga, K. Sadakane, and T. Sakai, editors, *String Processing and Information Retrieval: 23rd International Symposium, SPIRE 2016, Beppu, Japan, October 18-20, 2016, Proceedings*, pages 22–34, Cham, 2016. Springer International Publishing.

[CIK+16b] M. Crochemore, C. S. Iliopoulos, R. Kundu, M. Mohamed, and F. Vayani.
Linear algorithm for conservative degenerate pattern matching.
*Engineering Applications of Artificial Intelligence*, 51:109 – 114, 2016.

[CIK+17] M. Crochemore, C. S. Iliopoulos, T. Kociumaka, J. Radoszewski, W. Rytter, and T. Waleń.
Covering problems for partial words and for indeterminate strings.
*Theoretical Computer Science*, 698:25 – 39, 2017.
Algorithms, Strings and Theoretical Approaches in the Big Data Era (In Honor of the 60th Birthday of Professor Raffaele Giancarlo).

[CK16] P. Cording and M. Knudsen.
Maximal unbordered factors of random strings.
In S. Inenaga, K. Sadakane, and T. Sakai, editors, *Proceedings of the International Symposium String Processing and Information Retrieval SPIRE*, pages 93–96, 2016.

[CKML+16] K. Clark, I. Karsch-Mizrachi, D. J. Lipman, J. Ostell, and E. W. Sayers.
Genbank.
*Nucleic Acids Res*, 44(Database issue):D67–D72, Jan 2016.

[CR02] M. Crochemore and W. Rytter.
*Jewels of Stringology*.
World Scientific, 2002.

[Cro81] M. Crochemore.
An optimal algorithm for computing the repetitions in a word.
*Information Processing Letters*, 12(5):244 – 250, 1981.

[CS04] M. Crochemore and M.-F. Sagot.
Motifs in Sequences: Localization and Extraction.
In C. M. J. C. Konopka A. K., editor, *Compact Handbook of Computational Biology*, pages 47–97. Marcel Dekker, New York, 2004.

[CSS+15] D. M. Church, V. A. Schneider, K. M. Steinberg, M. C. Schatz, A. R. Quinlan, C.-S. Chin, P. A. Kitts, B. Aken, G. T. Marth, M. M. Hoffman, J. Herrero, M. L. Z. Mendoza, R. Durbin, and P. Flicek.
Extending reference assembly models.
*Genome Biology*, 16(1):13, 2015.

[dB46] N. G. de Bruijn.
A combinatorial problem.

*Koninklijke Nederlandse Akademie v. Wetenschappen*, 49:758–764, 1946.

[DCI+15] A. Dilthey, C. Cox, Z. Iqbal, M. R. Nelson, and G. McVean.
Improved genome inference in the MHC using a population reference graph.
*Nat Genet*, 47(6):682–688, Jun 2015.

[Dij97] E. W. Dijkstra.
*A Discipline of Programming*.
Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition, 1997.

[dKGC+11] A. P. J. de Koning, W. Gu, T. A. Castoe, M. A. Batzer, and D. D. Pollock.
Repetitive elements may comprise over two-thirds of the human genome.
*PLOS Genetics*, 7(12):1–12, 12 2011.

[DKK+04] J.-P. Duval, R. Kolpakov, G. Kucherov, T. Lecroq, and A. Lefebvre.
Linear-time computation of local periods.
*Theoretical Computer Science*, 326(1):229 – 240, 2004.

[DLL14] J.-P. Duval, T. Lecroq, and A. Lefebvre.
Linear computation of unbordered conjugate on unordered alphabet.
*Theoretical Computer Science*, 522:77 – 84, 2014.

[Dur13] S. Durocher.
A simple linear-space data structure for constant-time range minimum query.
In A. Brodnik, A. Løgpez-Ortiz, V. Raman, and A. Viola, editors, *Space-Efficient Data Structures, Streams, and Algorithms*, volume 8066 of *Lecture Notes in Computer Science*, pages 48–60. Springer Berlin Heidelberg, 2013.

[Duv82] J. Duval.
Relationship between the period of a finite word and the length of its unbordered segments.
*Discrete Mathematics*, 40(1):31 – 44, 1982.

[EP02] E. Eskin and P. A. Pevzner.

Finding composite regulatory patterns in dna sequences.
*Bioinformatics*, 18(suppl 1):S354–S363, 2002.

[ES79] A. Ehrenfeucht and D. Silberger.
Periodicity and unbordered segments of words.
*Discrete Math*, 26(2):101–109, 1979.

[FGB+17] I. Fontana, G. Giacalone, A. Bonanno, S. Mazzola, G. Basilone, S. Genovese, S. Aronica, S. Pissis, C. S. Iliopoulos, R. Kundu, A. Fiannaca, A. Langiu, G. L. Bosco, M. L. Rosa, and R. Rizzo.
Pelagic species identification by using a probabilistic neural network and echo-sounder data.
In A. Lintas, S. Rovetta, P. F. Verschure, and A. E. Villa, editors, *Artificial Neural Networks and Machine Learning – ICANN 2017: 26th International Conference on Artificial Neural Networks, Alghero, Italy, September 11-14, 2017, Proceedings*, volume 10613 LNCS of *Lecture Notes in Computer Science*, pages 454–455, Cham, 2017. Springer International Publishing.

[FH06] J. Fischer and V. Heun.
Theoretical and practical improvements on the RMQ- problem, with applications to LCA and LCE.
In M. Lewenstein and G. Valiente, editors, *Combinatorial Pattern Matching, 17th Annual Symposium, CPM 2006, Proceedings*, volume 4009 of *Lecture Notes in Computer Science*, pages 36–48. Springer, 2006.

[FP74] M. Fischer and M. Paterson.
*String-matching and Other Products*.
MAC technical memorandum. Mass. Inst. of Technology, Project MAC, 1974.

[FW90] M. L. Fredman and D. E. Willard.
Blasting through the information theoretic barrier with fusion trees.
In *Proceedings of the Twenty-second Annual ACM Symposium on Theory of Computing*, STOC '90, pages 1–7, New York, NY, USA, 1990. ACM.

[GIL+17] R. Grossi, C. S. Iliopoulos, C. Liu, N. Pisanti, S. P. Pissis, A. Retha, G. Rosone, F. Vayani, and L. Versari.
On-Line Pattern Matching on Similar Texts.
In J. Kärkkäinen, J. Radoszewski, and W. Rytter, editors, *28th Annual Symposium on Combinatorial Pattern Matching (CPM 2017)*, volume 78 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 9:1–9:14, Dagstuhl, Germany, 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[GKSS15] P. Gawrychowski, G. Kucherov, B. Sach, and T. Starikovskaya.
Computing the longest unbordered substring.
In C. Iliopoulos, S. Puglisi, and E. Yilmaz, editors, *Proceedings of the International Symposium String Processing and Information Retrieval SPIRE*, pages 246–257, 2015.

[GS18] GenBank and W. Statistics.
https://www.ncbi.nlm.nih.gov/genbank/statistics/, 2018.

[Gus97] D. Gusfield.
*Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*.
Cambridge University Press, New York, NY, USA, 1997.

[HL09] Y. Hou and S. Lin.
Distinct gene number-genome size relationships for eukaryotes and non-eukaryotes: Gene content estimation for dinoflagellate genomes.
*PLOS ONE*, 4(9):1–8, 09 2009.

[HN12] S. Holub and D. Nowotka.
The Ehrenfeucht–Silberger problem.
*Journal of Combinatorial Theory, Series A*, 119(3):668–682, 2012.

[HPB13] L. Huang, V. Popic, and S. Batzoglou.
Short read alignment with populations of genomes.
*Bioinformatics*, 29(13):i361–i370, 2013.

[HSW08] J. Holub, W. Smyth, and S. Wang.
Fast pattern-matching on indeterminate strings.

*Journal of Discrete Algorithms*, 6(1):37 – 50, 2008.
Selected papers from AWOCA 2005.

[HT84] D. Harel and R. Tarjan.
Fast algorithms for finding nearest common ancestors.
*SIAM Journal on Computing*, 13(2):338–355, 1984.

[IKM16] C. S. Iliopoulos, R. Kundu, and M. Mohamed.
Efficient computation of clustered-clumps in degenerate strings.
In L. Iliadis and I. Maglogiannis, editors, *Artificial Intelligence Applications and Innovations: 12th IFIP WG 12.5 International Conference and Workshops, AIAI 2016, Thessaloniki, Greece, September 16-18, 2016, Proceedings*, pages 510–519, Cham, 2016. Springer International Publishing.

[IKMV16] C. S. Iliopoulos, R. Kundu, M. Mohamed, and F. Vayani.
Popping superbubbles and discovering clumps: Recent developments in biological sequence analysis.
In M. Kaykobad and R. Petreschi, editors, *WALCOM: Algorithms and Computation: 10th International Workshop, WALCOM 2016, Kathmandu, Nepal, March 29–31, 2016, Proceedings*, pages 3–14, Cham, 2016. Springer International Publishing.

[IKP17] C. S. Iliopoulos, R. Kundu, and S. P. Pissis.
Efficient pattern matching in elastic-degenerate texts.
In F. Drewes, C. Martín-Vide, and B. Truthe, editors, *Language and Automata Theory and Applications: 11th International Conference, LATA 2017, Umeå, Sweden, March 6-9, 2017, Proceedings*, pages 131–142, Cham, 2017. Springer International Publishing.

[IMP96] C. S. Iliopoulos, D. W. G. Moore, and K. Park.
Covering a string.
*Algorithmica*, 16(3):288–297, Sep 1996.

[IMR08] C. S. Iliopoulos, L. Mouchard, and M. S. Rahman.
A new approach to pattern matching in degenerate dna/rna sequences and distributed pattern matching.
*Mathematics in Computer Science*, 1(4):557–569, Jun 2008.

[KKMP18] T. Kociumaka, R. Kundu, M. Mohamed, and S. P. Pissis.
Longest Unbordered Factor in Quasilinear Time.
In W.-L. Hsu, D.-T. Lee, and C.-S. Liao, editors, *29th International Symposium on Algorithms and Computation (ISAAC 2018)*, volume 123 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 70:1–70:13, Dagstuhl, Germany, 2018. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[KLA+01] T. Kasai, G. Lee, H. Arimura, S. Arikawa, and K. Park.
Linear-time longest-common-prefix computation in suffix arrays and its applications.
In *Proceedings of the Annual Symposium on Combinatorial Pattern Matching CPM*, pages 181–192, 2001.

[KLS15] G. Kucherov, A. Loptev, and T. Starikovskaya.
On maximal unbordered factors.
In *Proceedings of the Annual Symposium on Combinatorial Pattern Matching CPM*, Lecture Notes in Computer Science, pages 343–354. Springer, 2015.

[KM17] R. Kundu and T. Mahmoodi.
Mining acute stroke patients' data using supervised machine learning.
In J. Blömer, I. S. Kotsireas, T. Kutsia, and D. E. Simos, editors, *Mathematical Aspects of Computer and Information Sciences: 7th International Conference, MACIS 2017, Vienna, Austria, November 15-17, 2017, Proceedings*, pages 364–377, Cham, 2017. Springer International Publishing.

[KMP77] D. E. Knuth, J. H. Morris, and V. R. Pratt.
Fast pattern matching in strings.
*SIAM Journal on Computing*, 6(2):323–350, 1977.

[KPR16] T. Kociumaka, S. P. Pissis, and J. Radoszewski.
Pattern Matching and Consensus Problems on Weighted Sequences and Profiles.
In S.-H. Hong, editor, *27th International Symposium on Algorithms and Computation (ISAAC 2016)*, volume 64 of *Leibniz Inter-*

*national Proceedings in Informatics (LIPIcs)*, pages 46:1–46:12, Dagstuhl, Germany, 2016. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[KRRW12] T. Kociumaka, J. Radoszewski, W. Rytter, and T. Waleń.
Efficient data structures for the factor periodicity problem.
In *String Processing and Information Retrieval SPIRE*, pages 284–294. Springer, 2012.

[KRRW15] T. Kociumaka, J. Radoszewski, W. Rytter, and T. Walen.
Internal pattern matching queries in a text and applications.
In *Proceedings of the Annual ACM-SIAM Symposium on Discrete Algorithms SODA*, pages 532–551, 2015.

[LBKP14] Y. Li, J. Bailey, L. Kulik, and J. Pei.
Efficient matching of substrings in uncertain sequences.
In M. J. Zaki, Z. Obradovic, P. Tan, A. Banerjee, C. Kamath, and S. Parthasarathy, editors, *Proceedings of the 2014 SIAM International Conference on Data Mining, Philadelphia, Pennsylvania, USA, April 24-26, 2014*, pages 767–775. SIAM, 2014.

[LGN16] H. Lu, F. Giordano, and Z. Ning.
Oxford nanopore minion sequencing and genome assembly.
*Genomics, Proteomics & Bioinformatics*, 14(5):265 – 279, 2016.
SI: Big Data and Precision Medicine.

[LKM+14] Y. Liu, M. Koyutürk, S. Maxwell, M. Xiang, M. Veigl, R. S. Cooper, B. O. Tayo, L. Li, T. LaFramboise, Z. Wang, X. Zhu, and M. R. Chance.
Discovery of common sequences absent in the human reference genome using pooled samples from next generation sequencing.
*BMC Genomics*, 15(1):685, 2014.

[LLB+01] E. S. Lander, L. M. Linton, B. Birren, C. Nusbaum, M. C. Zody, J. Baldwin, K. Devon, K. Dewar, M. Doyle, W. FitzHugh, et al.
Initial sequencing and analysis of the human genome.
*Nature*, 409(6822):860–921, 2001.

[LV89] G. M. Landau and U. Vishkin.

Fast parallel and serial approximate string matching.
*J. Algorithms*, 10(2):157–169, June 1989.

[Mai89] M. G. Main.
Detecting leftmost maximal periodicities.
*Discrete Applied Mathematics*, 25(1):145 – 153, 1989.

[MBCT15] V. Mäkinen, D. Belazzougui, F. Cunial, and A. I. Tomescu.
*Genome-Scale Algorithms*, chapter 1.
Cambridge University Press, 2015.

[McC76] E. M. McCreight.
A space-economical suffix tree construction algorithm.
*Journal of the ACM (JACM)*, 23(2):262–272, 1976.

[MdOEMI16] S. Maciuca, C. del Ojo Elias, G. McVean, and Z. Iqbal.
*A Natural Encoding of Genetic Variation in a Burrows-Wheeler Transform to Enable Mapping and Genome Inference*, volume 9838, pages 222–233.
Springer International Publishing, Cham, 2016.

[MG77] A. M. Maxam and W. Gilbert.
A new method for sequencing dna.
*Proc Natl Acad Sci U S A*, 74(2):560–564, Feb 1977.

[MJP70] J. Morris Jr and V. Pratt.
A linear pattern-matching algorithm.
Technical Report 40, University of California, Berkeley, 1970.

[MM93] U. Manber and E. W. Myers.
Suffix arrays: A new method for on-line string searches.
*SIAM J. Comput.*, 22(5):935–948, 1993.

[MPVZ05] M. Morgante, A. Policriti, N. Vitacolonna, and A. Zuccolo.
Structured motifs search.
*J. Comput. Biol.*, 12(8):1065–1082, Oct 2005.

[OSS13] T. Onodera, K. Sadakane, and T. Shibuya.
Detecting superbubbles in assembly graphs.
In *WABI*, pages 338–348, 2013.

[Pis14] S. P. Pissis.
MoTeX-II: structured MoTif eXtraction from large-scale datasets.
*BMC Bioinformatics*, 15(1):235, 2014.

[PNEG17] B. Paten, A. M. Novak, J. M. Eizenga, and E. Garrison.
Genome graphs and the evolution of genome inference.
*Genome Res.*, 27(5):665–676, 05 2017.

[PNGH17] B. Paten, A. M. Novak, E. Garrison, and G. Hickey.
Superbubbles, ultrabubbles and cacti.
In S. C. Sahinalp, editor, *Research in Computational Molecular Biology*, pages 173–189, Cham, 2017. Springer International Publishing.

[PNH14] B. Paten, A. Novak, and D. Haussler.
Mapping to a Reference Genome Structure.
*ArXiv e-prints*, April 2014.

[PR18] S. P. Pissis and A. Retha.
Dictionary Matching in Elastic-Degenerate Texts with Applications in Searching VCF Files On-line.
In G. D'Angelo, editor, *17th International Symposium on Experimental Algorithms (SEA 2018)*, volume 103 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 16:1–16:14, Dagstuhl, Germany, 2018. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[PTW01] P. A. Pevzner, H. Tang, and M. S. Waterman.
An Eulerian path approach to DNA fragment assembly.
*Proceedings of the National Academy of Sciences of the U. S. A.*, 98(17):9748–9753, 2001.

[RIL+06] M. S. Rahman, C. S. Iliopoulos, I. Lee, M. Mohamed, and W. F. Smyth.
Finding patterns with variable length gaps or don't cares.
In *Computing and Combinatorics: 12th Annual International Conference, COCOON 2006, Taipei, Taiwan, August 15-18, 2006. Proceedings*, volume 4112, pages 146–155, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[Ruž08] M. Ružić.
Constructing efficient dictionaries in close to sorting time.
In L. Aceto, I. Damgård, L. A. Goldberg, M. M. Halldórsson, A. Ingólfsdóttir, and I. Walukiewicz, editors, *Automata, Languages and Programming, ICALP 2008, Part I*, volume 5125 of *LNCS*, pages 84–95. Springer, 2008.

[Sha81] M. Sharir.
A strong-connectivity algorithm and its applications in data flow analysis.
*Computers & Mathematics with Applications*, 7(1):67 – 72, 1981.

[Smy02] B. Smyth.
Special issue - computing patterns in strings: Foreword.
*Fundam. Inf.*, 56(1,2):.1–.2, October 2002.

[Smy13] W. Smyth.
Computing regularities in strings: A survey.
*European Journal of Combinatorics*, 34(1):3 – 14, 2013.
Combinatorics and Stringology.

[SNC77] F. Sanger, S. Nicklen, and A. R. Coulson.
Dna sequencing with chain-terminating inhibitors.
*Proc Natl Acad Sci U S A*, 74(12):5463–5467, Dec 1977.

[SSS+15] W. Sung, K. Sadakane, T. Shibuya, A. Belorkar, and I. Pyrogova.
An O(m log m)-time algorithm for detecting superbubbles.
*IEEE/ACM Trans. Comput. Biology Bioinform.*, 12(4):770–777, 2015.

[Sun90] D. M. Sunday.
A very fast substring search algorithm.
*Commun. ACM*, 33(8):132–142, August 1990.

[SV88] B. Schieber and U. Vishkin.
On finding lowest common ancestors: Simplification and parallelization.
*SIAM J. Comput.*, 17(6):1253–1262, December 1988.

[SvS05] J. A. Shapiro and R. von Sternberg.

Why repetitive DNA is essential to genome function.
*Biol Rev Camb Philos Soc*, 80(2):227–250, May 2005.

[SW09] W. F. SMYTH and S. WANG.
An adaptive hybrid pattern-matching algorithm on indeterminate strings.
*International Journal of Foundations of Computer Science*, 20(06):985–1004, 2009.

[Tar72] R. Tarjan.
Depth-first search and linear graph algorithms.
*SIAM Journal on Computing*, 1(2):146–160, 1972.

[Tar76] R. Tarjan.
Edge-disjoint spanning trees and depth-first search.
*Acta Informatica*, 6(2):171–185, 1976.

[THCS01] R. L. R. Thomas H. Cormen, Charles E. Leiserson and C. Stein.
*Introduction to Algorithms*.
MIT Press, Cambridge, MA., 2001.

[Ukk95] E. Ukkonen.
On-line construction of suffix trees.
*Algorithmica*, 14(3):249–260, 1995.

[VAM+01] J. C. Venter, M. D. Adams, E. W. Myers, P. W. Li, R. J. Mural, G. G. Sutton, H. O. Smith, M. Yandell, C. A. Evans, R. A. Holt, et al.
The sequence of the human genome.
*Science*, 291(5507):1304–1351, 2001.

[Wei73] P. Weiner.
Linear pattern matching algorithms.
In *Proceedings of the 14th IEEE Annual Symposium on Switching and Automata Theory*, pages 1–11. Institute of Electrical Electronics Engineer, 1973.

[WM92] S. Wu and U. Manber.
Fast text searching: Allowing errors.
*Commun. ACM*, 35(10):83–91, October 1992.

[WPL+16] T. Woods, T. Preeprem, K. Lee, W. Chang, and B. Vidakovic.
Characterizing exons and introns by regularity of nucleotide strings.
*Biology Direct*, 11(1):6, Feb 2016.

[WQX14] X. Wu, J.-P. Qiang, and F. Xie.
Pattern matching with flexible wildcards.
*Journal of Computer Science and Technology*, 29(5):740–750, Sep 2014.

[ZB08] D. R. Zerbino and E. Birney.
Velvet: algorithms for de novo short read assembly using de Bruijn graphs.
*Genome Research*, 18(5):821–829, 2008.

[ZGI08] H. Zhang, Q. Guo, and C. S. Iliopoulos.
Generalized approximate regularities in strings.
*International Journal of Computer Mathematics*, 85(2):155–168, 2008.